NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ON DISTRIBUTED STRATEGIES IN DEFENSE OF A HIGH VALUE UNIT (HVU) AGAINST A SWARM ATTACK

by

Sze Yi Ding

September 2012

Thesis Co-Advisors:	Isaac Kaminer Johannes O. Royset

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2012	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE On Distributed Strategies in Defense of a High Value Unit (HVU) Against a Swarm Attack	**5. FUNDING NUMBERS**
6. AUTHOR(S) Sze Yi Ding	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government .______N/A________

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Swarm attacks are of great concern to the U.S. Navy as well as to navies around the world and commercial ships transiting through waters with high volume of marine traffic. A large group of hostile ships can hide themselves among various other small ships, like pleasure crafts, fishing boats and transport vessels, and can make a coordinated attack against a High Value Unit (HVU) while it passes by. The HVU can easily be overwhelmed by the numbers and sustain heavy damage or risk being taken over.

The objective of this thesis is to develop heuristic algorithms that multiple defenders can use to intercept and stop the advances of multiple attackers. The attackers are in much larger numbers compared to the defenders, and are moving in on a slow moving HVU. Pursuit guidance laws and proportional navigation (PN) guidance laws, commonly used in missile guidance strategies, are modified to be used by the defenders to try intercepting attackers that outnumber them.

Another objective is to evaluate the effectiveness of the heuristic algorithms in defending the HVU against the swarm attack. The probability that the HVU survives the swarm attack will be used as a measure of effectiveness of the algorithms. The impact of various parameters, like the number of defenders and the speed of defenders, on the effectiveness of the algorithms are also evaluated.

14. SUBJECT TERMS Swarm Attack, High Value Unit	**15. NUMBER OF PAGES** 105
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ON DISTRIBUTED STRATEGIES IN DEFENSE OF A HIGH VALUE UNIT (HVU) AGAINST A SWARM ATTACK

Sze Yi Ding
DSO National Laboratories (Singapore)
B.S.M.E., National University of Singapore, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2012

Author: Sze Yi Ding

Approved by: Isaac Kaminer
Thesis Co-Advisor

Johannes O. Royset
Thesis Co-Advisor

Knox T. Millsaps
Chair, Department of Mechanical and Aerospace Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Swarm attacks are of great concern to the U.S. Navy as well as to navies around the world and commercial ships transiting through waters with high volume of marine traffic. A large group of hostile ships can hide themselves among various other small ships, like pleasure crafts, fishing boats and transport vessels, and can make a coordinated attack against a High Value Unit (HVU) while it passes by. The HVU can easily be overwhelmed by the numbers and sustain heavy damage or risk being taken over.

The objective of this thesis is to develop heuristic algorithms that multiple defenders can use to intercept and stop the advances of multiple attackers. The attackers are in much larger numbers compared to the defenders, and are moving in on a slow moving HVU. Pursuit guidance laws and proportional navigation (PN) guidance laws, commonly used in missile guidance strategies, are modified to be used by the defenders to try intercepting attackers that outnumber them.

Another objective is to evaluate the effectiveness of the heuristic algorithms in defending the HVU against the swarm attack. The probability that the HVU survives the swarm attack will be used as a measure of effectiveness of the algorithms. The impact of various parameters, like the number of defenders and the speed of defenders, on the effectiveness of the algorithms are also evaluated.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
- A. PROBLEM STATEMENT 1
- B. OBJECTIVES 1
- C. LITERATURE REVIEW 2
- D. THESIS ORGANIZATION 2

II. METHODOLOGY 5
- A. SCENARIO 5
- B. SIMULATION 5
  - 1. High Value Unit (HVU) 6
  - 2. Attackers 7
  - 3. Defenders 8
  - 4. Cost Function 9
- C. SIMULATION PARAMETERS 15
  - 1. Guidance Laws 15
    - *a. Pursuit Guidance 15*
    - *b. PN Guidance 17*
  - 2. Number of Defenders 19
  - 3. Speed Ratio 20
  - 4. Test Matrix 20

III. RESULTS ANALYSIS 23
- A. SIMULATION RESULTS 23
  - 1. Guidance Strategies 23
  - 2. Number of Defenders 28
  - 3. Speed Ratio 30

IV. CONCLUSION AND RECOMMENDED FUTURE WORK 35
- A. CONCLUSION 35
- B. RECOMMENDED FUTURE WORK 35
  - 1. Other Guidance Methods or Strategies 35
  - 2. Different Engagement Scenarios 36
  - 3. Intent Recognition 36
  - 4. Hit Rate Function Modeling 37

APPENDIX A. MATLAB® SIMULATION CODE 39
- A. SIMULATION RUN SCRIPT FILES 39
- B. ATTACKER & DEFENDER TRAJECTORY GENERATION CODE 43
- C. PERFORMANCE RELATED FUNCTIONS 57

LIST OF REFERENCES 85

INITIAL DISTRIBUTION LIST 87

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. Motion of High Value Unit 7
Figure 2. Attackers trajectories 8
Figure 3. Defenders trajectories to intercept attackers 9
Figure 4. Hit rate function modeled using beta function in 3D 12
Figure 5. Hit rate of k-th defender against attacker 13
Figure 6. Angularly decaying rate function reflecting FOV limitations 14
Figure 7. Pursuit guidance with weightage on LOS vectors 17
Figure 8. Proportional Navigation guidance with weighted attacker centroid 19
Figure 9. Comparison of pursuit guidance and PN guidance 24
Figure 10. Typical engagement scenario using pursuit guidance 25
Figure 11. Pursuit guidance engagement scenario 26
Figure 12. Typical engagement scenario using PN guidance 27
Figure 13. Proportional Navigation guidance engagement scenario 28
Figure 14. Comparison for varying number of defenders 29
Figure 15. Comparison for varying speed ratios using pursuit and PN guidance 32
Figure 16. Effect of small but finite values of hit rate function on qlt 33

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Scenarios to be simulated 22
Table 2. Measure of Effectiveness for pursuit guidance and PN guidance 24
Table 3. Measure of Effectiveness for varying number of defenders................ 29
Table 4. Measure of Effectiveness for pursuit guidance and PN guidance with varying speed ratios 31

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

FOV	Field of View
HVU	High Value Unit
LOS	Line-of-Sight
MOE	Measure of Effectiveness
PN	Proportional Navigation
UCSC	University of California, Santa Cruz
USV	Unmanned Surface Vehicles

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The author wishes to thank Professor Isaac Kaminer for his guidance and assistance in the completion of this thesis. Many thanks go to Professor Johannes Royset for his insights and suggestions on how the thesis can progress. Thanks to Claire Walton from UCSC for developing the algorithm and cost function to evaluate the performance of the heuristic strategies used.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

Since the attack on U.S.S. Cole in 2000 and the U.S.-Iranian naval dispute in the Strait of Hormuz in 2008, awareness for asymmetric warfare has increased. Swarm attacks, in particular, are of great concern to the U.S. Navy as well as to navies around the world and commercial ships transiting through waters with high volume of marine traffic. A large group of hostile ships can hide themselves among various other small ships, like pleasure crafts, fishing boats and transport vessels, and can make a coordinated attack against a High Value Unit (HVU) while it passes by. The HVU can easily be overwhelmed by the numbers and sustain heavy damage or risk being taken over.

The NPS thesis by Tiwari, 2008 [1] presented the capability gap in defending against such a swarm attack. This thesis aims to study defensive strategies that can be used to defend against swarm attacks, and evaluates the mission effectiveness of these strategies using a cost function that can realistically represent a multi-agent engagement scenario.

B. OBJECTIVES

The objective of this thesis is to develop heuristic algorithms that multiple defenders can use to intercept and stop the advances of multiple attackers. The attackers are in much larger numbers compared to the defenders, and are moving in on a slow moving HVU. Pursuit guidance laws and proportional navigation (PN) guidance laws, commonly used in missile applications, are modified to be used by the defenders to try intercepting attackers that outnumbers them.

Another objective is to evaluate the effectiveness of the heuristic algorithms in defending the HVU against the swarm attack. The probability that the HVU survives the swarm attack will be used as a measure of effectiveness of the algorithms. The impact of various parameters, like the number of defenders and the speed of defenders, on the effectiveness of the algorithms are also evaluated.

C. LITERATURE REVIEW

Various existing literature related to multi-agent engagement scenarios usually focus on search algorithms. Chung et al, 2011 [2] looks at optimal detection of underwater intruder in a channel using Unmanned Underwater Vehicles. Royset & Sato, 2010 [3] considers route optimization for multiple searchers to look for one or more probabilistically moving target(s). In these two papers, there is only one target with multiple defenders searching for it. Jang & Tomlin, 2005 [4], Shaferman & Oshman, 2009 [5], Shin, 2011 [6] investigates various cooperative guidance strategies in multi-agent engagement scenarios, but are mostly limited to missile guidance and considers scenarios with only 3 or 4 targets, defended with an equal number of defenders. Rozen, 2009 [7], studies the detection, recognition and interception of multiple targets using an interdiction force of a UAV and a navy vessel, but the number of targets are small and moves at random and cannot be considered as a swarm. In addition, mission effectiveness is based only on the number of targets considered intercepted under conditions that are not reflective of real life situations.

This thesis explores strategies that naval ships can use to engage attackers that greatly outnumbers themselves effectively, and uses a cost function that considers the hit rate of the defenders against the attackers and the attackers against the HVU, to work out the mission effectiveness by evaluating the survival rate of the HVU.

D. THESIS ORGANIZATION

In the next chapter, the setup of the simulation is explained, describing the parameters used to simulate the motion of the HVU, the attackers and the defenders. The derivation of the cost function used to evaluate the mission effectiveness of the algorithm will also be shown. A test matrix is formulated to investigate the effects of adjusting various parameters have on the defense strategies. Analysis of the results obtained is presented in Chapter III, followed by the conclusion and recommendations for future work that can be done in this area of research in Chapter IV.

THIS PAGE INTENTIONALLY LEFT BLANK

II. METHODOLOGY

A. SCENARIO

Consider a HVU, like an aircraft carrier or a supply container ship, moving through waters with a high number of small ships in the region. Some of these small ships may have no hostile intent, but hidden amongst them are a group of ships that are intending to attack the HVU. When the HVU passes by their location, these hostile ships can launch a coordinated attack against the HVU. They maneuver towards HVU at the same time, and in large numbers. This is what we consider as a swarm attack. The attackers may be equipped with small caliber guns and some may carry explosive with the intent of conducting a suicide attack. The HVU detects these attackers on its radar and determines that it is under imminent attack based on the number of contacts moving towards its location at high speed. The HVU then deploys multiple Unmanned Surface Vehicles (USVs) equipped with weaponry, to intercept and neutralize these attackers before they can get close enough to become a threat. The number of USVs (defenders) deployed is small compared to the number of attackers.

B. SIMULATION

The above scenario is simulated using MATLAB®. The setup of the scenarios to be studied is to be based upon real life platforms and weapons systems to give a realistic model and therefore a representative simulation of actual hostile engagement scenarios. Current Fast Attack Crafts (FAC) and Fast Inshore Attack Crafts (FIAC) have max speeds in excess of 40 knots (~20 m/s). As such, the attackers are modeled to have max speeds of 45 knots (~23 m/s), while varying top speeds for defenders between 20 knots (~10m/s) to 60knots (~30 m/s) will be investigated. The hostile crafts are assumed to be armed with small caliber guns, or carrying explosives for suicidal attacks. The maximum effective ranges of these types of weapons do not exceed 1 nmi (~1.8 km). Therefore, to be able to intercept the hostile targets before they get within effective range of their weapons, the defenders have to intercept these hostile targets at least 1nmi from the HVU. At the lowest defender speed we will be investigating, the defenders will take about 3 minutes to travel 1 nmi. During this time, the attackers can travel upwards of 2.25 nmi. In other words, the defenders have to be deployed to intercept these hostile targets at least 3.25 nmi from the HVU. For the simulations, the attackers will start at 4.25 nmi (~8 km) away from the HVU to provide a margin of safety. Modern sensors systems carried by Naval Fighting Ships can detect targets as far away as 100 to 200 km. So detecting these targets would not be a problem at all. The weapon systems carried by the defenders will be small to medium caliber guns having max effective ranges from 1nmi (~1800 m) to about 2 nmi (~3700 m).

A cost function is developed that will allow us to evaluate the effectiveness of the defense strategies by calculating the probability of the HVU surviving the swarm attack. We can vary some of the parameters used in the scenario to look at how some of these parameters affect the survival rate of the HVU. Some of the parameters we are going to look at are, guidance laws used by the defenders to track the attackers, the number of defenders and the speed ratio of the defenders to attackers.

The following paragraphs will explain how the simulation is set up to simulate the motions of the HVU, attackers and defenders, as well as the describing the guidance laws used by the defenders. Following which, a matrix of scenarios is formed to explore the effects of varying parameters in the simulation.

1. High Value Unit (HVU)

The HVU is simulated to be a relatively slow moving vessel with limited maneuvering capabilities. In the simulation, the HVU will start at the origin (0, 0), and moves at a constant velocity of 5 m/s in a straight line directly to the north (in the positive y direction). The HVU is assumed to have detected the hostile targets at a distance away and is starting to deploy its defensive force of USVs. The HVU will not be taking evasive actions to avoid the attackers in this simulation, since it cannot outmaneuver the attackers anyway. The HVU has to depend on the defensive force and onboard defensive capabilities to fend off the attack.

Figure 1 shows a plot of the HVU motion in two-dimensional space over the total simulation time, T.

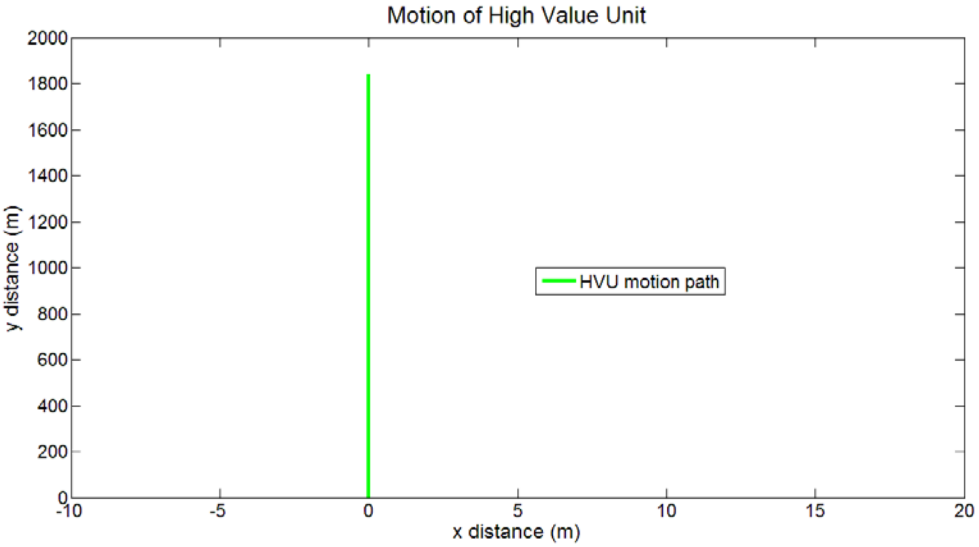


Figure 1. Motion of High Value Unit

2. Attackers

The initial positions of the 40 attackers are randomly generated with uniform distribution over a rectangular area to the east of the HVU (positive x-direction). This area spans from x = 8000 to x = 8100, and y = 0 to y = 500. The attackers' trajectories are generated using a time-coordinated path following control architecture described in Ghabcheloo et al, 2009[8], with their speeds constrained to a maximum of 23m/s. The trajectories thus generated are collision-free, and ensures that the attackers reach the HVU at the same time for a coordinated attack. The attacker are assumed to ignore the defenders as they try to intercept them, as they will be focused on the task at hand to destroy the HVU using a suicidal attack.

The weapons carried on these attacking crafts will be limited to small and medium caliber guns, as well as explosives, all of which having effective ranges of no more than 1800m.

Figure 2 shows the typical trajectories of 40 attackers in a coordinate attack against the HVU.

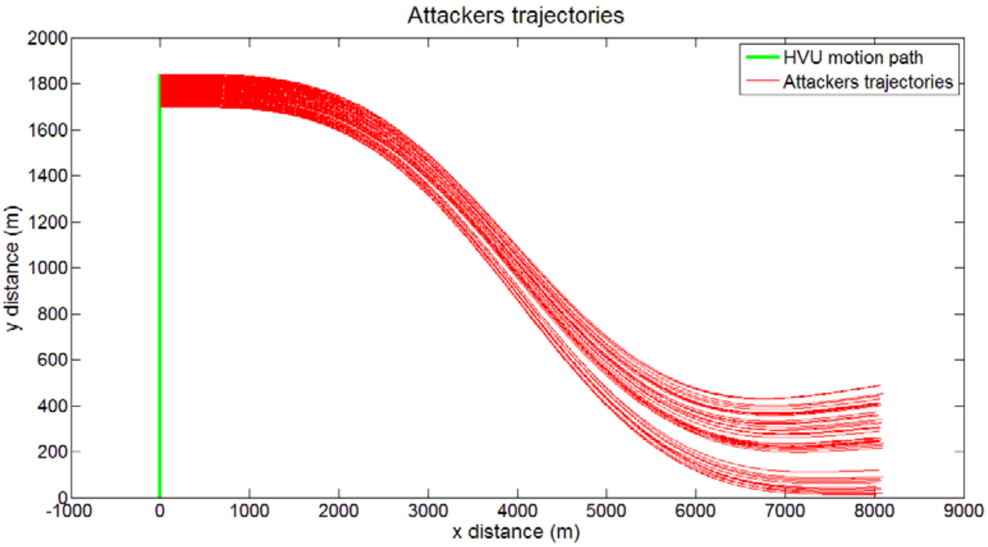


Figure 2. Attackers trajectories

3. Defenders

The defenders are simulated to be moving in formation alongside the HVU, and are deployed simultaneously to intercept the incoming attackers. They will start moving towards the attackers at their maximum speeds as soon as they are deployed. A divide and conquer tactic will be used, where each defender will be assigned a group of attackers to intercept and neutralize. The defenders will use either modified PN guidance law or pursuit guidance law to guide them towards the group of assigned attackers. The guidance laws used will be discussed in more detail later in this chapter.

The field of view (FOV) for these defenders will be ±60^o from their heading. When an attacker is within the defenders' FOV and weapons' effective range, the defenders will shoot at the attackers in an attempt to neutralize it. If there are multiple attackers within the defender's sight, the defender will divide its attention evenly to each of these attackers that are in sight. This is to ensure that the defender can neutralize as many attackers as it can before the attackers maneuvers past it.

Figure 3 shows a typical engagement scenario where five defenders try to intercept 40 attackers that are coming to attack the HVU.

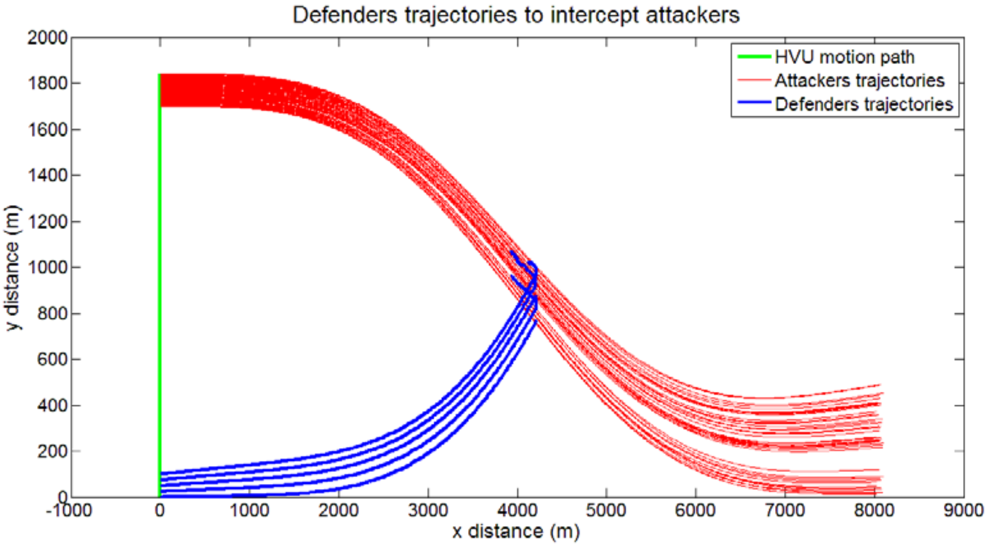


Figure 3. Defenders trajectories to intercept attackers

4. Cost Function

The effectiveness of the defense strategy is evaluated by estimating the probability of the HVU surviving the swarm attack. A cost function, developed by Claire Walton from UCSC [9], derives the probability that the HVU has survived until time $t + \Delta t$ using conditional probabilities. This probability can be represented as

$$p(t + \Delta t) = p(t) \cdot f(t, \Delta t) \qquad (2.1)$$

where $f(t, \Delta t)$ is the probability that the HVU survived in the time interval $[t, t + \Delta t]$. By modeling the function $f(t, \Delta t)$, a differential equation for $p(t)$ can be obtained by taking the limit, $\Delta t \to 0$.

To formulate the function $f(t, \Delta t)$, let us first consider a single defender protecting a HVU from a single attacker. Let $p(t)$ be the probability that the HVU survived until time t. The goal is to maximize the value of $p(T)$ at the end of the simulation when $t = T$.

Let:

- $q(t) =$ the probability that the attacker survived to time t
- $s_a(t) =$ instantaneous attacker hit rate against HVU
- $s_d(t) =$ instantaneous defender hit rate against attacker

The probability of the attacker surviving until time $t + \Delta t$ can be expressed as

$$q(t + \Delta t) = q(t)(1 - s_d(t)\Delta t) \qquad (2.2)$$

This creates the differential equation

$$\dot{q}(t) = -q(t)s_d(t) \qquad (2.3)$$

Solving this equation yields

$$q(t) = e^{-\int_0^t s_d(\tau)d\tau} \qquad (2.4)$$

The effective attackers hit rate against the HVU at time t, given that the attacker survived until that time, is therefore $q(t)s_a(t)$. The probability that the HVU surviving until time $t + \Delta t$ can then be expressed in a similar fashion to $q(t)$,

$$p(t + \Delta t) = p(t)(1 - q(t)s_a(t)\Delta t) \qquad (2.5)$$

which yields

$$p(t) = e^{-\int_0^t q(\tau)s_a(\tau)d\tau} \qquad (2.6)$$

Substitution of $q(t)$ into this expression and evaluating to time T, we obtain the cost function,

$$p(T) = e^{-\int_0^T e^{-\int_0^T s_d(\omega)d\omega} s_a(\tau)d\tau} \qquad (2.7)$$

To expand this to a multi-agent engagement scenario, the same method can be applied, but the hit rate of each defender against each attacker has to be considered. To do this, we define the following terms

- $q_l(t)$ = the probability that the attacker l survived to time t
- $s_{a,l}(t)$ = l-th attacker hit rate against HVU
- $s_{d,k}(t)$ = k-th defender hit rate against attacker

Now, the probability that the l-th attacker survives until time t is

$$q_l(t + \Delta t) = q_l(t)\prod_{k=1}^{K}\left(1 - s_{d,k}(t)\Delta t\right) \qquad (2.7)$$

this yields

$$q_l(t) = e^{-\int_0^t \sum_{k=1}^{K} s_{d,k}(\tau)d\tau} \qquad (2.8)$$

Similarly, we write for the HVU,

$$p(t + \Delta t) = p(t)\prod_{l=1}^{L}\left(1 - q_l(t)s_{a,l}(t)\Delta t\right) \qquad (2.9)$$

This can be made into a differential equation if we expand the product and discard higher order terms of Δt,

$$p(t + \Delta t) = p(t) + \left[\left(1 - \sum_{l=1}^{L} q_l(t)s_{a,l}(t)\Delta t + h.o.t.\right) - 1\right]p(t) \qquad (2.10)$$

$$p(t + \Delta t) = p(t) + \left[-\sum_{l=1}^{L} q_l(t)s_{a,l}(t)\Delta t + h.o.t.\right]p(t) \qquad (2.11)$$

The differential equation can then be written

$$\dot{p}(t) = \left(-\sum_{l=1}^{L} q_l(t) s_{a,l}(t)\right) p(t) \tag{2.12}$$

and the solution evaluated to time T is

$$p(T) = e^{-\int_0^T \left(\sum_{l=1}^{L} q_l(t) s_{a,l}(t)\right) d\tau} \tag{2.13}$$

The l-th attacker hit rate against the HVU is modeled using a beta function, with positive parameters $\alpha, \beta \geq 2$. The function can be connected smoothly (C^1) as a piecewise function,

$$s_{a,l}\big(r(t)\big) = \begin{cases} r(t)^{\alpha}\big(1 - r(t)\big)^{\beta} & , 0 \leq r(t) \leq 1 \\ 0 & , else \end{cases} \tag{2.14}$$

where $r(t)$ is the range of the attacker to the HVU. The parameters are adjusted to reflect the hit rate of the attackers' weaponry against the HVU. In the simulations, the range of this hit rate is limited to 800m. Figure 4 shows an example of the hit rate function modeled with the beta function.

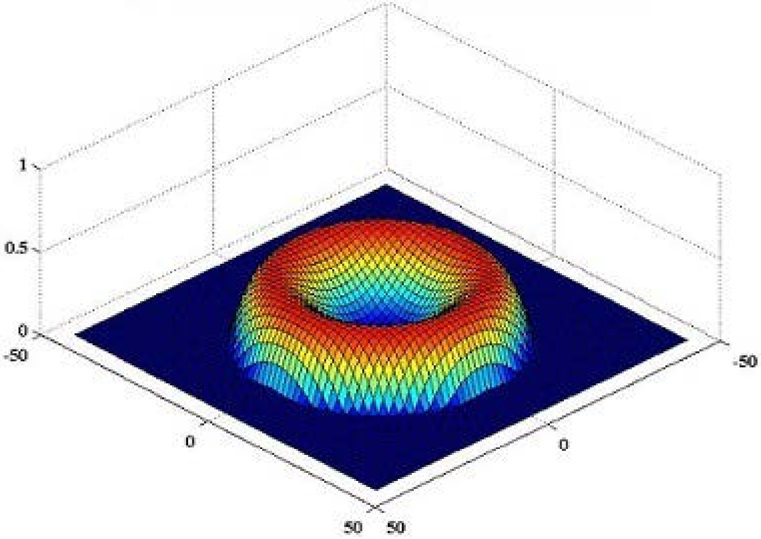


Figure 4. Hit rate function modeled using beta function in 3D

As for the defenders' hit rate against the attackers, the weapons carried by the defenders are assumed to be small to medium caliber guns. The maximum effective ranges of such weapons are expected to be around 1800m to 3700m. The peak effectiveness of the weapon is expected to occur at a range of around 100 to 200m from the defender. This is because if the attacker is any closer to the defender, the LOS rate to the attacker is going to be very high, especially for a cross-range engagement scenario. Limitations of the swiveling rate of the guns will make it hard to keep up with the moving attackers. The hit rate will also decrease from the peak value at 100–200m at an exponential rate. This reflects the fact that the lethality of the shells fired at the attackers reduces with increasing distance travelled. Accuracy of the shells also decreases with increasing range due to wind and weapon recoil. Figure 5 shows the expected hit rate function.

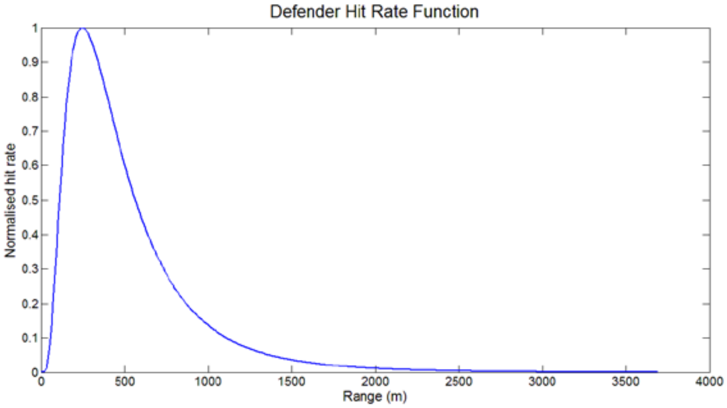


Figure 5. Hit rate of k-th defender against attacker

The hit rate function shown here is modeled with a lognormal probability distribution curve. Note that the hit rate function modeled this way does not have a finite range like the beta functions have. The value decreases to a very small, but finite value at very large range. This leads to some unexpected results, which will be discussed later in the results section.

To simulate the limited FOV of the weapon system, the hit rate function is modified with an angularly decaying multiplier, which decreases with arc angle.

Figure 6 shows the multiplier function with such FOV limitations in three-dimensional space.

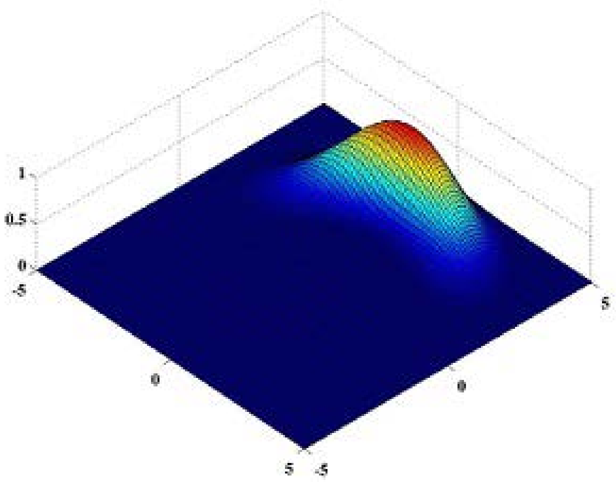


Figure 6. Angularly decaying rate function reflecting FOV limitations

Another situation the hit rate function should account for is the fact that when multiple attackers are within a single defender's FOV, the defender's attention is divided between the attackers. This division of attention can be simulated by dividing the hit rate function by another function that smoothly approximates the number of attackers in its FOV. We can define

- x_k = the location of the k-th defender
- x_l = the location of the l-th attacker

The following sum of Gaussian distributions can then be used as the division function,

$$\sum_{l=1}^{L} \Phi\left(\frac{\rho - \|x_k - x_l\|}{\sigma}\right) \qquad (2.15)$$

with the standard deviation σ set to a small number, to approximate the number of attackers within a radius ρ of the k-th defender.

C. SIMULATION PARAMETERS

In this section, we look at the various parameters in the simulation that can influence the effectiveness of the defensive force. We will look at three parameters, the guidance strategy the defenders use to intercept the attackers, the number of defenders deployed and the speed ratio of the defenders' speed to the attackers' speed.

1. Guidance Laws

Here, we are going to look at two guidance laws. One is based on pursuit guidance and the other is based on proportional navigation (PN) guidance. Pursuit guidance and PN guidance are common guidance laws used in missile guidance. Zarchan [10] describes these guidance laws in detail. Modern missiles use more complex forms of guidance laws, but in this thesis, we are only going to look at the basic pursuit and PN guidance laws for a start. In a nutshell, pursuit guidance generates turn commands that points the defender to the line-of-sight (LOS) to the attacker, while PN guidance generates turn commands proportional to LOS rate to form an intercept triangle to intercept the attacker along its path of motion. Both of LOS angle and LOS rate can be easily obtained from EO seekers that can be equipped on a USV.

a. Pursuit Guidance

In the simulation, pursuit guidance is implemented by having a defender change its heading to face directly at the attacker. This is akin to giving a turn command that will turn the defender an amount equal to the LOS angle in a single time step. This simplifies calculation and implementation, as we only need to find the LOS vector. When the defender is up against a swarm of attackers, the pursuit guidance law is not going to work in its basic form, as we have multiple LOS vectors instead of just one. Therefore, for the defender to use the guidance law, all the LOS vectors are combined into one "effective" vector that the defender will use. This combination can be achieved by finding a weighted sum of the LOS vectors. We consider that in a real life engagement scenario, priority should be given to attackers that the defenders can reach first.

In the simulation, we have the following information

- $\boldsymbol{P_d} = \begin{bmatrix} x_d \\ y_d \end{bmatrix}$ = the position of the defender
- $\boldsymbol{P_{a,l}} = \begin{bmatrix} x_{a,l} \\ y_{a,l} \end{bmatrix}$ = the position of the l-th attacker
- $\boldsymbol{V_d} = \begin{bmatrix} \dot{x}_d \\ \dot{y}_d \end{bmatrix}$ = the velocity of the defender
- $\boldsymbol{V_{a,l}} = \begin{bmatrix} \dot{x}_{a,l} \\ \dot{y}_{a,l} \end{bmatrix}$ = the velocity of the l-th attacker

From this we can obtain

- d_l, the distance of the defender to the l-th attacker

$$d_l = \left\| \boldsymbol{P_d} - \boldsymbol{P_{a,l}} \right\| \quad (2.16)$$

- $\boldsymbol{i_{a,l}}$**,** the unit LOS vector of the defender to the l-th attacker

$$\boldsymbol{i_{a,l}} = \left(\boldsymbol{P_d} - \boldsymbol{P_{a,l}} \right) / d_l \quad (2.17)$$

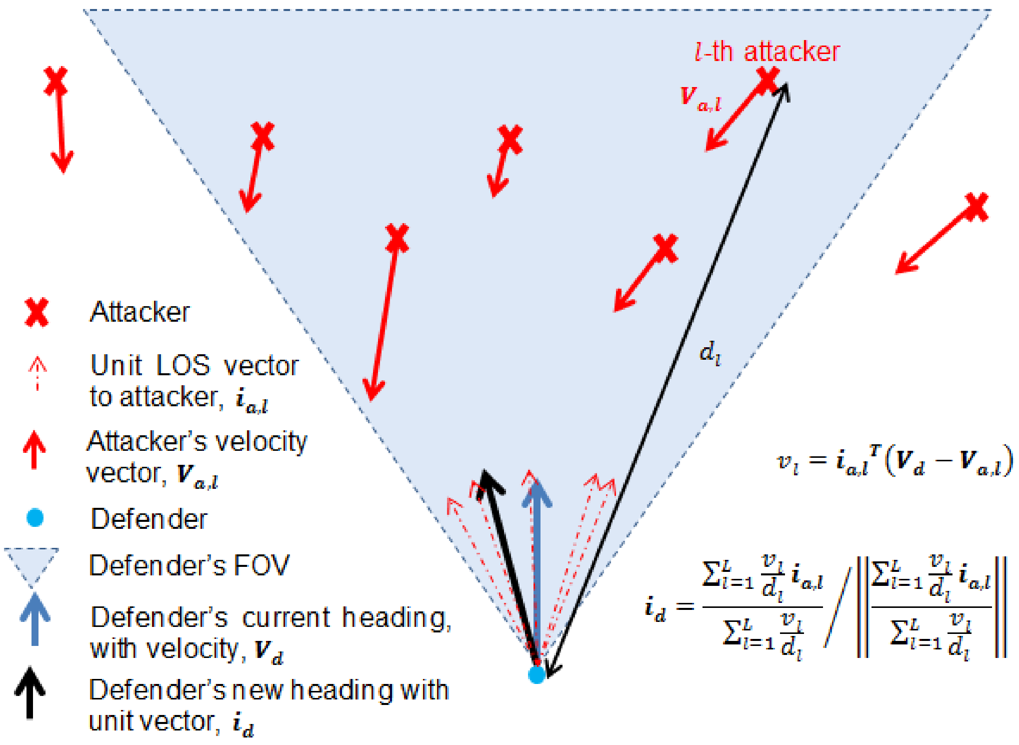
- v_j is the closing velocity of the defender to the l-th attacker

$$v_l = \boldsymbol{i_{a,l}}^{\boldsymbol{T}} \left(\boldsymbol{V_d} - \boldsymbol{V_{a,l}} \right) \quad (2.18)$$

Therefore, the unit velocity vector of the defender,

$$\boldsymbol{i_d} = \frac{\sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{i_{a,l}}}{\sum_{l=1}^{L} \frac{v_l}{d_l}} \Bigg/ \left\| \frac{\sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{i_{a,l}}}{\sum_{l=1}^{L} \frac{v_l}{d_l}} \right\| \quad (2.19)$$

This places priority on the attacker that the defender can reach in the shortest time, since $\frac{v_l}{d_l} = \frac{1}{t_l}$ where t_l is the time to reach the l-th attacker based on a straight-line intercept course of the defender to the attacker along the LOS vector. The lower the time, t_l is, the higher the weightage will be for the l-th attacker. Figure 7 illustrates this method.



l-th attacker
$\boldsymbol{V}_{a,l}$
d_l
$v_l = \boldsymbol{i}_{a,l}{}^T(\boldsymbol{V}_d - \boldsymbol{V}_{a,l})$
$\boldsymbol{i}_d = \frac{\sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{i}_{a,l}}{\sum_{l=1}^{L} \frac{v_l}{d_l}} \Bigg/ \left\| \frac{\sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{i}_{a,l}}{\sum_{l=1}^{L} \frac{v_l}{d_l}} \right\|$
Attacker
Unit LOS vector to attacker, $\boldsymbol{i}_{a,l}$
Attacker's velocity vector, $\boldsymbol{V}_{a,l}$
Defender
Defender's FOV
Defender's current heading, with velocity, $\boldsymbol{V}_d$
Defender's new heading with unit vector, $\boldsymbol{i}_d$

Figure 7. Pursuit guidance with weightage on LOS vectors

Note that only the attackers that are within the FOV of the defenders, as denoted by the shaded triangle in the figure, are included in the computation.

b. PN Guidance

For PN guidance, a turn command is generated that is proportional to the LOS rate of the attacker. This puts the defenders on an intercept triangle where the defender will intercept the attacker along its path of motion. As we have multiple attackers in this case, we need to come up with a single turn command for the defender such that it will track the general swarm movement, while giving priority to the closest threat. One possible way to implement PN guidance for multiple attackers is to compute the centroid of the attacker in the defender's FOV, as well as the effective velocity of that centroid, and apply PN guidance to intercept this moving centroid. This centroid is to be weighted by the closing velocity and distance to each attacker to give priority to attackers with lower intercept times. To do this we can find

The centroid of attackers in defender's FOV,

$$\boldsymbol{P_a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix} = \sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{P_{a,l}} \Bigg/ \sum_{l=1}^{L} \frac{v_l}{d_l} \tag{2.20}$$

and the velocity of this centroid,

$$\boldsymbol{V_a} = \begin{bmatrix} \dot{x}_a \\ \dot{y}_a \end{bmatrix} = \sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{V_{a,l}} \Bigg/ \sum_{l=1}^{L} \frac{v_l}{d_l} \tag{2.21}$$

From which, we find

- d_a, the distance of the defender to the attacker centroid

$$d_a = \|\boldsymbol{P_d} - \boldsymbol{P_a}\| \tag{2.22}$$

- $\boldsymbol{i_a}$**,** the unit LOS vector of the defender to the attacker centroid

$$\boldsymbol{i_a} = \begin{bmatrix} \cos\theta_a \\ \sin\theta_a \end{bmatrix} = (\boldsymbol{P_d} - \boldsymbol{P_a})/d_a \tag{2.23}$$

Now, we can obtain the LOS rate by

$$\dot{\theta}_a = \frac{(\dot{x}_d - \dot{x}_a)\sin\theta_a - (\dot{y}_d - \dot{y}_a)\cos\theta_a}{d_a} \tag{2.24}$$

The turn command needed, ω_c, is then calculated by

$$\omega_c = K\dot{\theta}_a \tag{2.25}$$

with the gain of the PN guidance law, $K = 6$

Figure 8 illustrates this method.

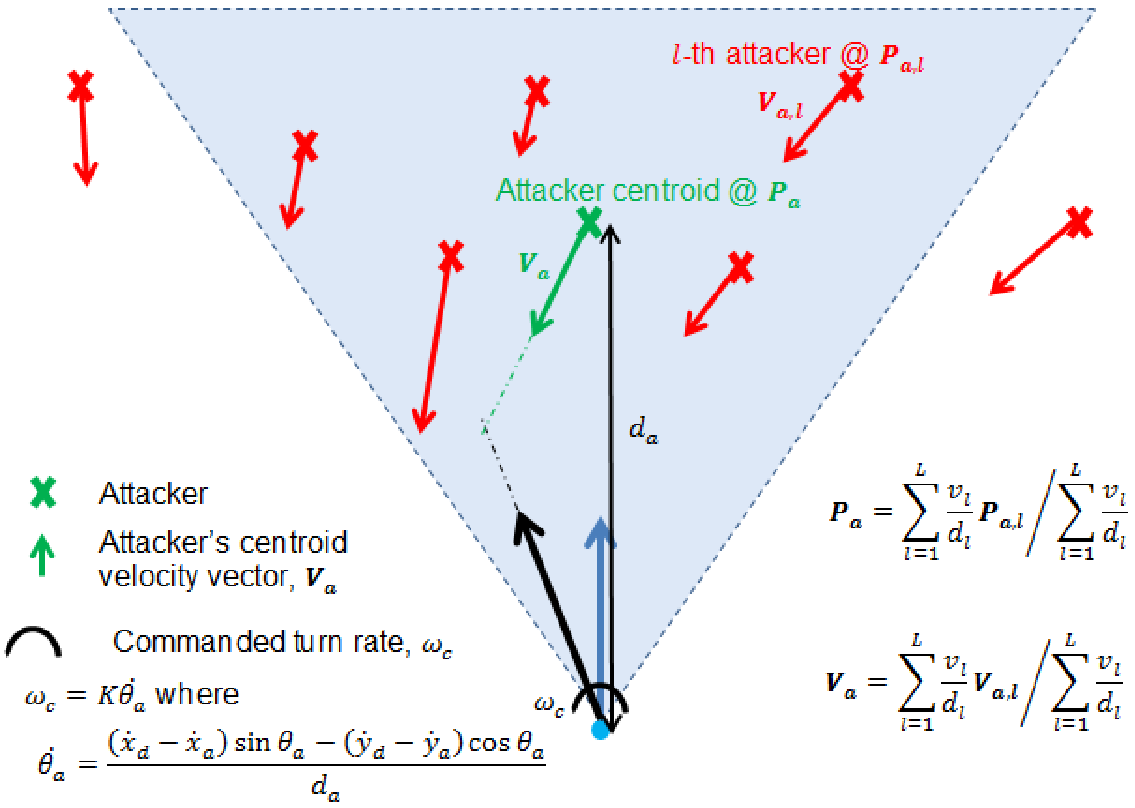

l-th attacker @ $\boldsymbol{P}_{a,l}$
$\boldsymbol{V}_{a,l}$
Attacker centroid @ $\boldsymbol{P}_a$
$\boldsymbol{V}_a$
d_a
Attacker
Attacker's centroid velocity vector, $\boldsymbol{V}_a$
Commanded turn rate, ω_c
$\omega_c = K\dot{\theta}_a$ where
$\dot{\theta}_a = \frac{(\dot{x}_d - \dot{x}_a)\sin\theta_a - (\dot{y}_d - \dot{y}_a)\cos\theta_a}{d_a}$
ω_c
$\boldsymbol{P}_a = \sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{P}_{a,l} \Big/ \sum_{l=1}^{L} \frac{v_l}{d_l}$
$\boldsymbol{V}_a = \sum_{l=1}^{L} \frac{v_l}{d_l} \boldsymbol{V}_{a,l} \Big/ \sum_{l=1}^{L} \frac{v_l}{d_l}$

Figure 8. Proportional Navigation guidance with weighted attacker centroid

Again, note that only attackers within the defender's FOV is included in the computation of the required turn rate.

2. Number of Defenders

By increasing the number of defenders we have against a swarm, we can expect the defenders to take down more attackers before they can get to the HVU. Hence, the chances of survival of the HVU can be expected to improve with increasing number of defenders. However, we can also expect that there would be some sort of diminishing return, where at some point, increasing the number of defenders is not going to increase the HVU survival rate significantly. The number of defenders deployed will be varied from 1 to 25.

3. Speed Ratio

The speed of the defenders will determine where they can first intercept the incoming attackers. It also affects the amount of time that the defenders have on the target. We would not want the defenders to intercept the attackers too late. Otherwise, they can be close enough to the HVU to pose a threat. However, we would not want the defenders to be moving so fast that they cannot target the attackers long enough to take them down. The speeds of the defenders will be varied from 10m/s to 30 m/s (about 19knots to 58knots), while the attackers' speed will be kept at 23 m/s (45knots). This corresponds to speed ratios between 0.4348 and 1.304.

4. Test Matrix

In each of these scenarios, 50 randomly generated attacker trajectories will be evaluated to obtain an average performance of the defense strategy used. This is to make sure that the performance of the defense strategy is not a limited to a specific attacker trajectory. The MATLAB® source code written for the simulation is documented in Appendix A of this thesis.

Table 1 summarizes the scenarios that are simulated.

Parameter	Number of defenders	Number of attackers	Speed of defenders (m/s)	Speed of attackers (m/s)	Guidance Law used
Guidance Law	1	40	25	23	Pursuit
	2	40	25	23	Pursuit
	3	40	25	23	Pursuit
	4	40	25	23	Pursuit
	5	40	25	23	Pursuit
	1	40	25	23	PN
	2	40	25	23	PN
	3	40	25	23	PN
	4	40	25	23	PN
	5	40	25	23	PN
# of Defenders	6	40	25	23	PN
	7	40	25	23	PN
	8	40	25	23	PN
	9	40	25	23	PN
	10	40	25	23	PN
	15	40	25	23	PN
	20	40	25	23	PN
	25	40	25	23	PN
Speed Ratio	5	40	10	23	PN
	5	40	11	23	PN
	5	40	12	23	PN
	5	40	13	23	PN
	5	40	14	23	PN
	5	40	15	23	PN
	5	40	16	23	PN
	5	40	17	23	PN
	5	40	18	23	PN
	5	40	19	23	PN
	5	40	20	23	PN
	5	40	21	23	PN
	5	40	22	23	PN
	5	40	23	23	PN
	5	40	24	23	PN
	5	40	25	23	PN
	5	40	30	23	PN

Parameter	Number of defenders	Number of attackers	Speed of defenders (m/s)	Speed of attackers (m/s)	Guidance Law used
Speed Ratio	5	40	10	23	Pursuit
	5	40	11	23	Pursuit
	5	40	12	23	Pursuit
	5	40	13	23	Pursuit
	5	40	14	23	Pursuit
	5	40	15	23	Pursuit
	5	40	16	23	Pursuit
	5	40	17	23	Pursuit
	5	40	18	23	Pursuit
	5	40	19	23	Pursuit
	5	40	20	23	Pursuit
	5	40	21	23	Pursuit
	5	40	22	23	Pursuit
	5	40	23	23	Pursuit
	5	40	24	23	Pursuit
	5	40	25	23	Pursuit
	5	40	30	23	Pursuit

Table 1. Scenarios to be simulated

III. RESULTS ANALYSIS

A. SIMULATION RESULTS

The measure of effectiveness (MOE) used to evaluate the performance of the defense strategy used is the probability of the HVU surviving the swarm attack at the end of the simulation, $p(T)$. The larger the objective value is, the more effective the defense strategy is. The values that $p(T)$ can take are between 0 and 1, with $p(T) = 0$ meaning that the HVU is guaranteed to be destroyed in the swarm attack and $p(T) = 1$ meaning that the HVU is guaranteed to survive the swarm attack.

In this chapter, we look at the performance of various strategies with varying parameters as described in Chapter II Section C, and analyze whether the simulation is sufficiently describing the engagement scenario. In all scenarios simulated, there will be 40 attackers, each having a maximum speed of 23m/s. 50 sets of initial attacker positions are generated randomly using uniform distribution over an area from x = 8000 to x = 8100, and y = 0 to y = 500. This corresponds to 50 different sets of attacker trajectory. The simulation will be repeated using the same 50 sets of attacker position for each of the cases to be simulated, and the mean of the MOE obtained from the 50 simulations will be used as the basis for comparison between cases.

1. Guidance Strategies

The performance of pursuit guidance is compared to that of PN guidance, with the number of defenders between 1 and 5. The speed of the defenders is kept at 25m/s, corresponding to a speed ratio of 1.1. The slight advantage in speed for the defenders is to allow the guidance laws to perform better against the attackers. Table 2 summarizes the objective values obtained and Figure 9 plots the values for comparison.

Parameter	Number of defenders	Guidance Law used	MOE	
			Mean	Standard deviation
Guidance Law	1	Pursuit	0.000314	0.002160
	2	Pursuit	0.092827	0.041404
	3	Pursuit	0.939744	0.011297
	4	Pursuit	0.997371	0.000407
	5	Pursuit	0.999848	0.000029
	1	PN	0.000000	0.000000
	2	PN	0.074510	0.057751
	3	PN	0.171997	0.282923
	4	PN	0.222126	0.294244
	5	PN	0.239250	0.301488

Table 2. Measure of Effectiveness for pursuit guidance and PN guidance

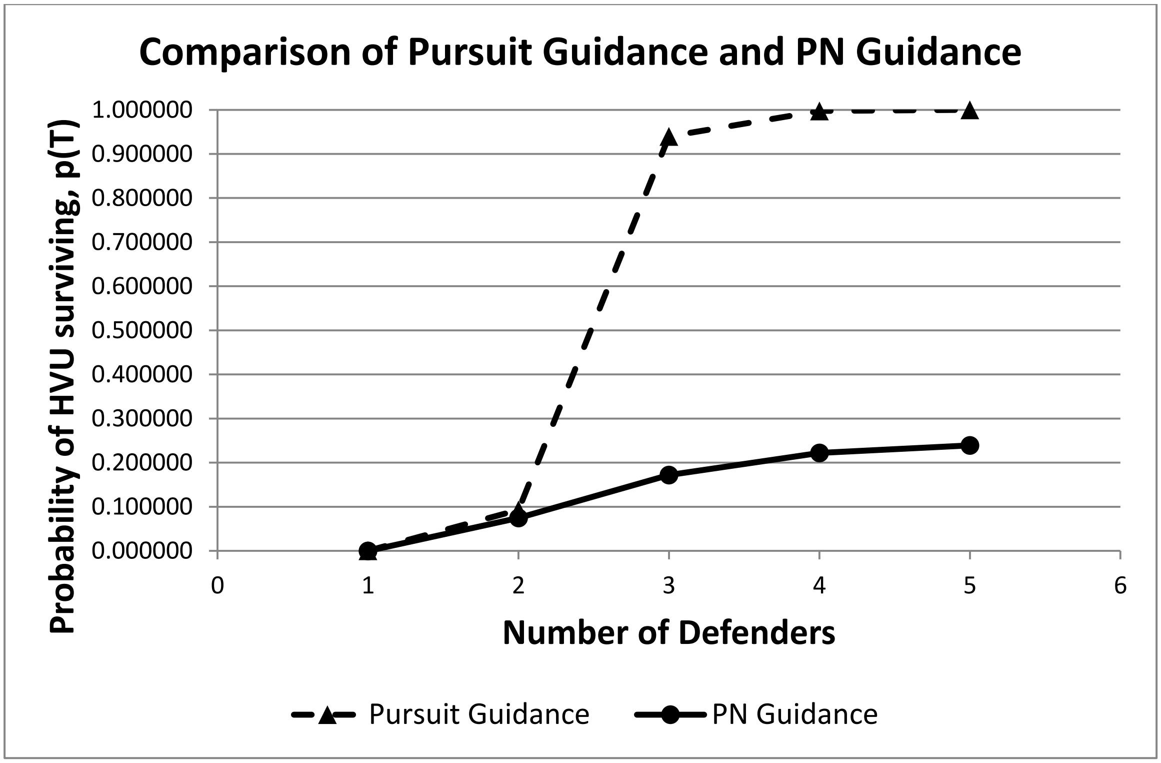


Figure 9. Comparison of pursuit guidance and PN guidance

The comparison between pursuit guidance and PN guidance yields a rather surprising result. Pursuit guidance, which is usually considered inferior to PN guidance in missile applications, seems to perform better, needing only three defenders to obtain a MOE of about 0.94. In other words, the HVU survives 94% of the time when three defenders are used against a swarm of 40 attackers. Comparatively, PN guidance can only achieve a MOE of about 0.24 with five defenders, meaning the HVU survives only about 24% of the time. The reason for this result is obvious when we look at the trajectory of the defenders. Figure 10 shows a typical engagement scenario when pursuit guidance is used.

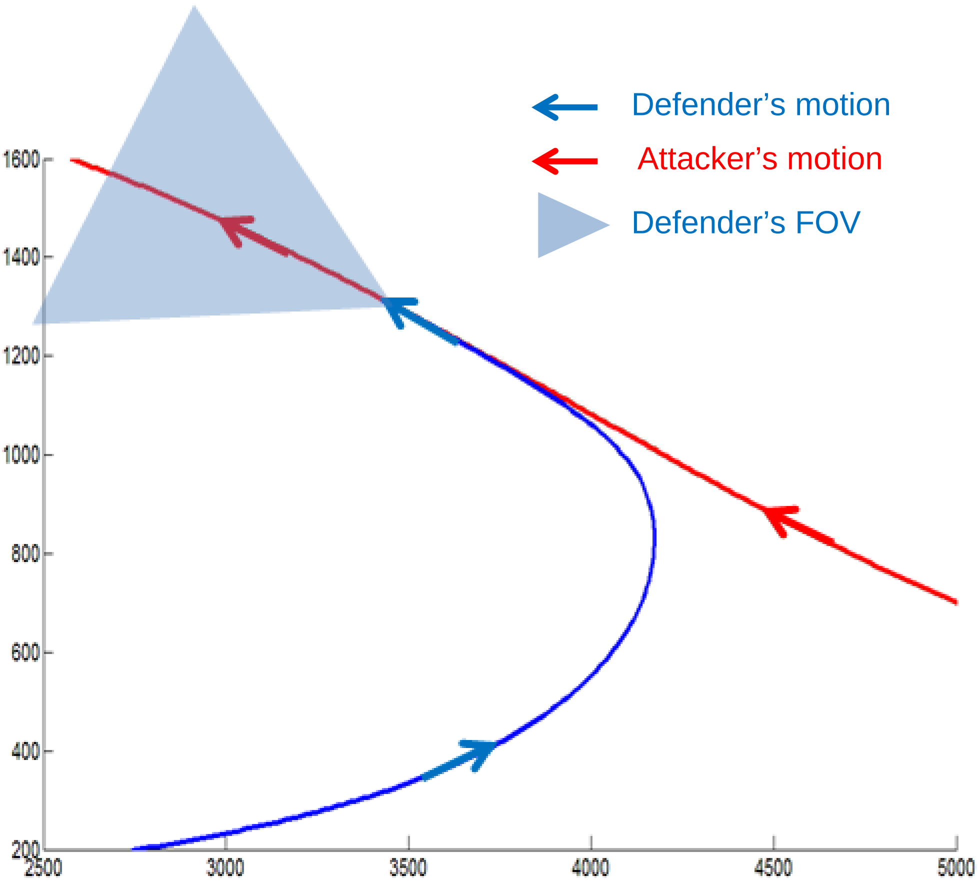


Figure 10. Typical engagement scenario using pursuit guidance

As can be seen in the engagement scenario, the defenders will always end up in a tail chase against the attackers. This is to be expected when pursuit guidance is used. However, unlike missile engagement scenarios, the probability of neutralizing the attackers does not depend on how close the defenders can get to the attackers, but rather on the time on target that the defenders have. Once the defenders get behind the attackers, the attackers are effectively constantly within the defenders' FOV and attacking range. Therefore, the probability of the attackers being neutralized increases greatly.

Figure 11 shows the engagement scenario where five defenders are deployed against a swarm of 40 attackers using pursuit guidance. It can be seen from the plot that the defenders were able to neutralize all 40 attackers before they can reach the HVU by coming up behind them and shooting them down.

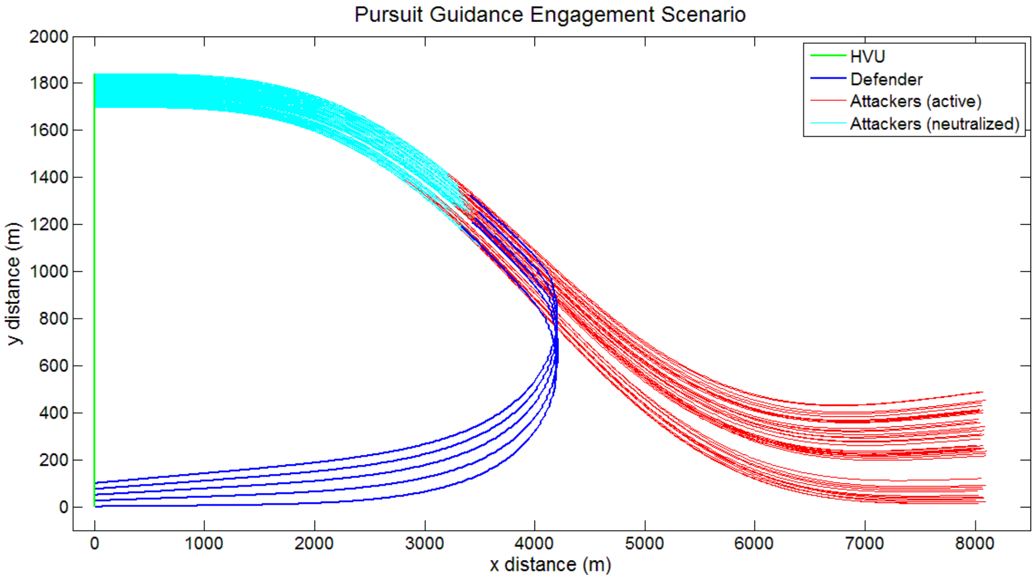


Figure 11. Pursuit guidance engagement scenario

PN guidance, on the other hand, gives rise to an engagement scenario where the attackers are going to pass in front of the defenders and out of their FOV in a relatively short period. Since the defenders will stop once no attackers are within their FOV, they will not give chase to the attackers from behind. This limits the amount of time the defenders have in dealing with the attackers, and subsequently results in a smaller chance of neutralizing the attackers. A typical engagement scenario using PN guidance is shown in Figure 12.

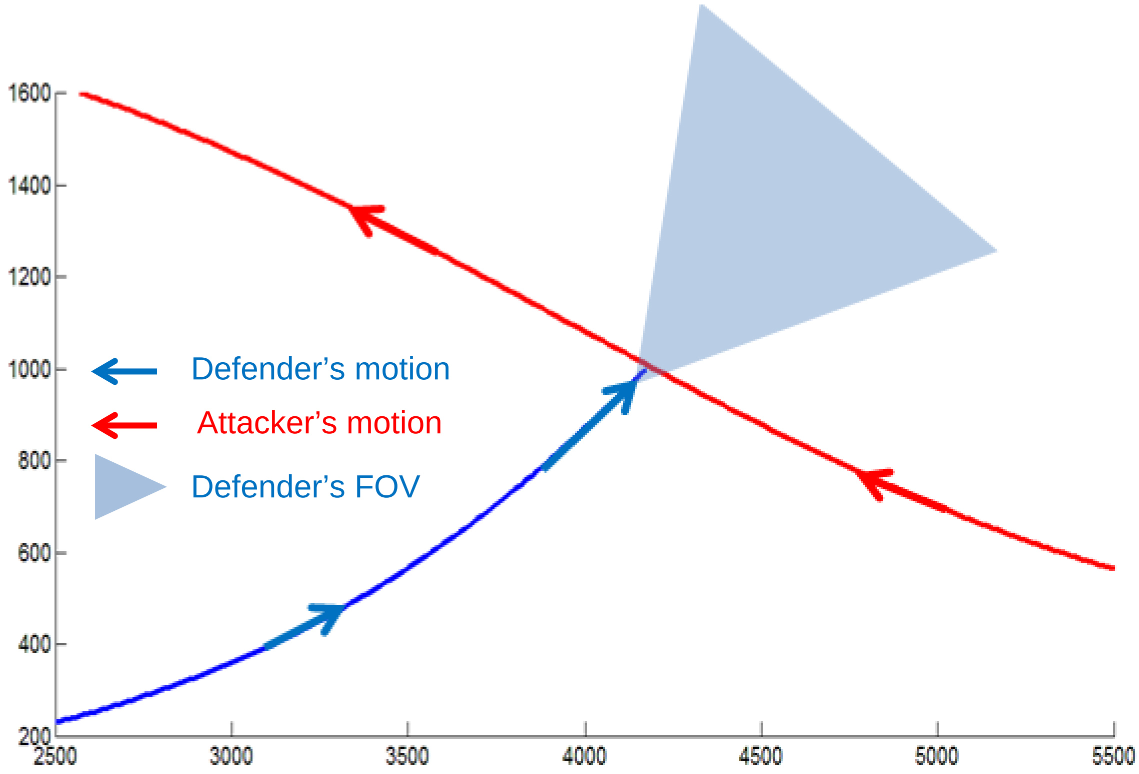


Figure 12. Typical engagement scenario using PN guidance

We can see from this engagement scenario that if attackers are able to get pass the defenders, they can reach the HVU without any hindrance, resulting in high probabilities of HVU being destroyed. Figure 13 shows the engagement scenario where five defenders are deployed against a swarm of 40 attackers using PN guidance. The scenario shows that some of the attackers were able to get past the defenders because of the short time period they were passing into the FOV of the defenders.

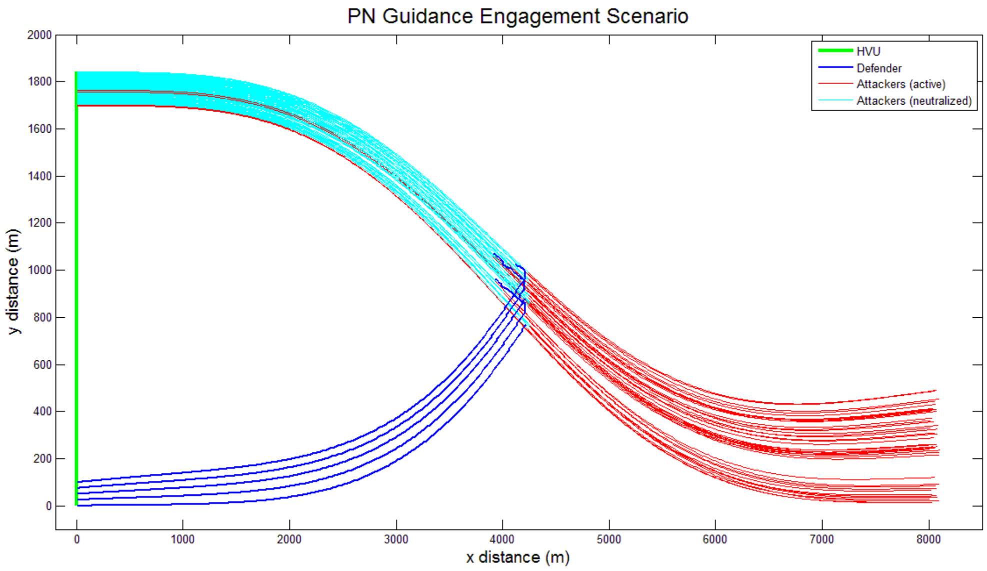


Figure 13. Proportional Navigation guidance engagement scenario

2. Number of Defenders

When the number of defenders is increased, each defender will have fewer attackers to deal with. Hence, the defenders should be able to neutralize the attackers with greater ease. To compare the effects of increasing the number of defenders, we vary the number of defenders from 1 to 25. The speed ratio is kept at 1.1 for each of the simulation run. Table 3 summarizes the objective values obtained and Figure 14 plots the values for comparison.

The plot shows that increasing the number of defenders does indeed improve the chances that the HVU survives. One observation here is that the MOE increases almost proportionally with the increase of defenders, up to 10 defenders when the probability of HVU surviving increases up to around 0.94. Increasing the number of defenders beyond 10 gives diminishing returns, with the probability increasing slightly to 0.9978 with 15 defenders (a 99.78% chance of HVU surviving). We can deduce from this result that each defender can take down four attackers efficiently in this particular engagement scenario. Any additional defenders may just mean that defenders are idling after defeating their own share of attackers, thus reducing the utility of these defenders.

Parameter	Number of defenders	MOE	
		Mean	Standard deviation
# of Defenders	1	0.000000	0.000000
	2	0.074510	0.057751
	3	0.171997	0.282923
	4	0.222126	0.294244
	5	0.239250	0.301488
	6	0.434544	0.334362
	7	0.497450	0.314897
	8	0.698302	0.213622
	9	0.827713	0.134149
	10	0.935115	0.065392
	15	0.997801	0.001932
	20	0.999950	0.000038
	25	0.999998	0.000002

Table 3. Measure of Effectiveness for varying number of defenders

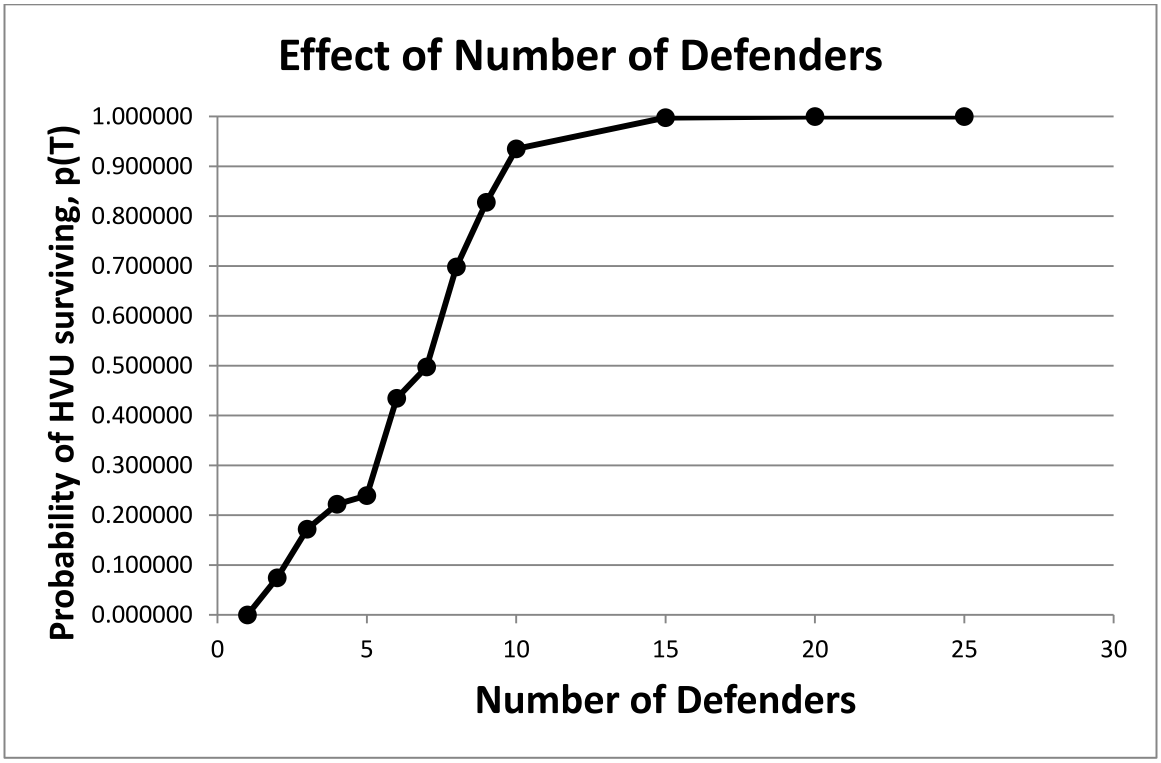


Figure 14. Comparison for varying number of defenders

3. Speed Ratio

The speed of the defenders is now varied between 10m/s and 30m/s. With the attackers' speed at 25m/s, this corresponds to speed ratios between 0.4348 and 1.3043. Five defenders are deployed in each case. Both pursuit guidance and PN guidance are evaluated for their effectiveness at the different speed ratios. Table 4 summarizes the objective values obtained and Figure 15 plots the values for comparison.

The MOE when PN guidance is used does not seem to have a significant difference, statistically speaking, when the speed ratio is above 0.74. However, the defenders generally seemed to perform better at lower speed ratios between 0.65 and 0.70. A probable reasoning behind this is that at lower speed ratios, the attackers remain within the defenders' FOV for a longer time, allowing longer dwell times for the defenders to neutralize the attackers. Having said that though, the attackers are expected to be intercepted and neutralized at a distance closer to the HVU when the speed ratio is lower. This means the safety margin is smaller and the HVU may come under fire if the attackers carry weapons with longer range. At even lower speed ratios, we can see that the performance decreases dramatically. One possible reason is that at lower speeds, the defenders have to aim at a point far ahead of the attacker to be able to intercept the attackers using PN guidance. If they do that though, the attackers may be at the edge of their FOV, lowering their hit rate against the attackers. The attackers may be even out of the FOV of the attackers if the speed of the defender is low enough, and the defenders will be unable to shoot at the attackers.

On the other hand, when pursuit guidance is used, a surprising result is obtained. The MOE for simulated speed ratios between about 0.4348 and 1.304 when pursuit guidance is used is either 1 or close to one. This means that the HVU is almost guaranteed to survive regardless of the speed of the defenders. Additional simulations are run at speeds of 5m/s, 1m/s and 0.01m/s, corresponding to speed ratio of 0.2174, 0.0435 and 0.0004 respectively.

Parameter	Guidance Law used	Speed of Defenders (m/s)	Speed Ratio	MOE	
				Mean	Standard Deviation
Speed Ratio	PN	10	0.4348	0.000	0.000
		11	0.4783	0.000	0.000
		12	0.5217	0.000	0.000
		13	0.5652	0.000	0.000
		14	0.6087	0.020	0.004
		15	0.6522	0.675	0.326
		16	0.6957	0.571	0.377
		17	0.7391	0.230	0.317
		18	0.7826	0.246	0.300
		19	0.8261	0.168	0.265
		20	0.8696	0.283	0.295
		21	0.9130	0.257	0.269
		22	0.9565	0.344	0.302
		23	1.0000	0.350	0.316
		24	1.0435	0.207	0.258
		25	1.0870	0.239	0.301
		30	1.3043	0.144	0.230
	Pursuit	0.01	0.0004	0.188	0.007
		1	0.0435	0.872	0.004
		5	0.2174	1.000	0.000
		10	0.4348	1.000	0.000
		11	0.4783	1.000	0.000
		12	0.5217	1.000	0.000
		13	0.5652	1.000	0.000
		14	0.6087	1.000	0.000
		15	0.6522	1.000	0.000
		16	0.6957	0.996	0.000
		17	0.7391	0.996	0.000
		18	0.7826	0.999	0.000
		19	0.8261	1.000	0.000
		20	0.8696	1.000	0.000
		21	0.9130	1.000	0.000
		22	0.9565	1.000	0.000
		23	1.0000	1.000	0.000
		24	1.0435	1.000	0.000
		25	1.0870	1.000	0.000
		30	1.3043	0.973	0.125

Table 4. Measure of Effectiveness for pursuit guidance and PN guidance with varying speed ratios

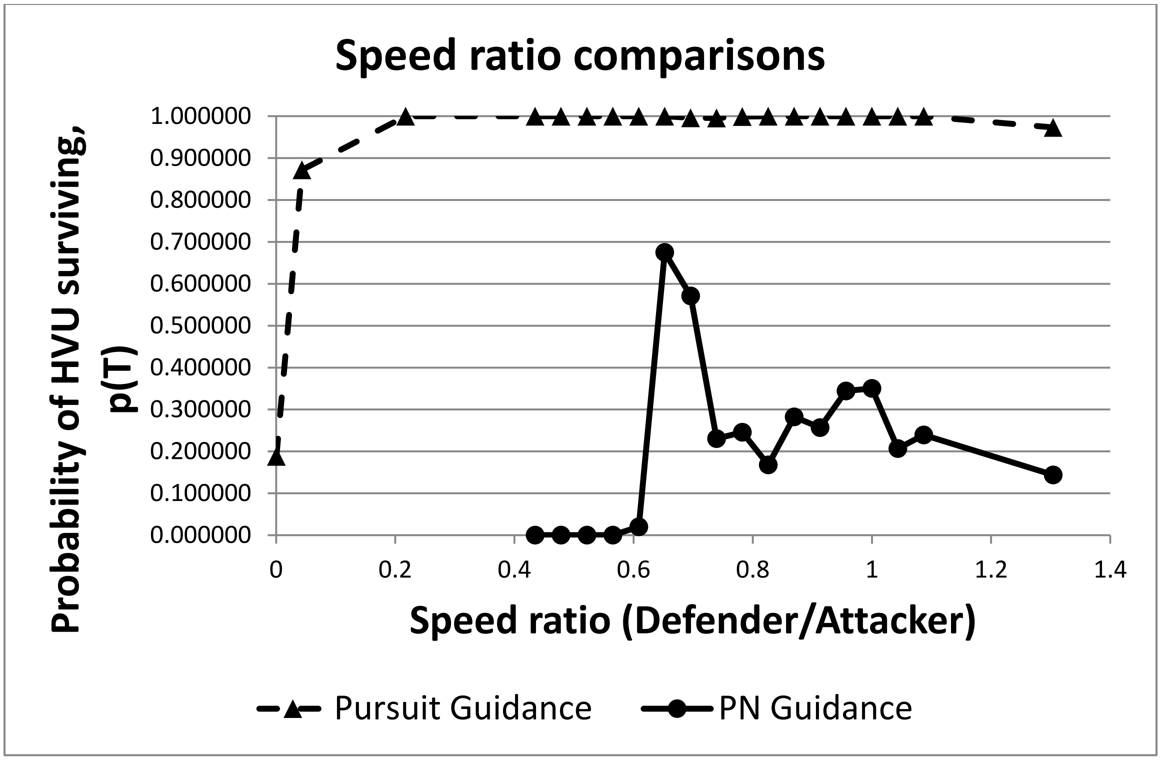


Figure 15. Comparison for varying speed ratios using pursuit and PN guidance

This is done to observe whether the MOE will remain high when the defenders are almost stationary. The MOE only started decreasing at a speed ratio of 0.0435. Even at an almost stationary speed of 0.01m/s, the MOE is at 0.188. What this means is that even if the defenders are barely moving, as long as they used pursuit guidance against the attackers, the HVU can still survive with an 18.8% chance.

The reason that we get such unrealistic MOE for pursuit guidance is the way the defenders hit rate against the attackers is modeled. As described in Chapter II section B.4, the hit rate function for the attackers is modeled with a lognormal function. The way the lognormal function is shaped is such that the value decreases gradually from a peak value in an exponential way. However, the value does not go to zero even at very large values. In other words, there is a small but finite value for the hit rate function that is being integrated by the part of the cost function that calculates the survival rate of the attackers. Since pursuit guidance will always point the defender in the direction where the attackers are, this small but finite value is constantly causing the probability of the attackers surviving ($q_l(t)$) to decrease. Figure 16 illustrates this situation where the probability that the attackers survives is constantly decreasing even when they should be out of the weapon range of the defenders.

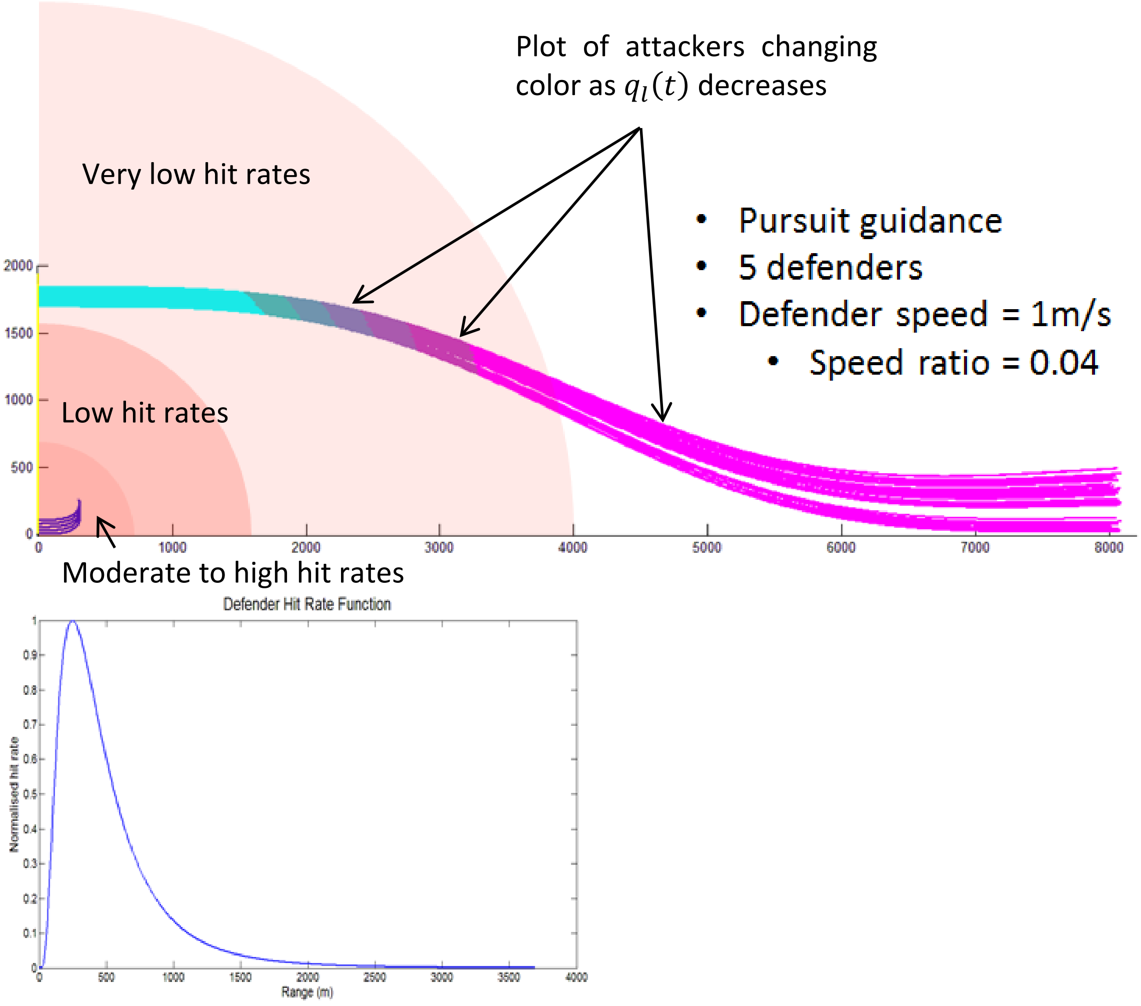


Figure 16. Effect of small but finite values of hit rate function on $q_l(t)$

We can see from the trajectory of the attackers that the attackers almost always remain within a zone where the hit rate of the defenders against them is very low. However, since the defenders are barely moving in this case, the closing speed of the attackers is lower than when the defenders are moving at faster speeds. Hence, the time that the attackers are within the defenders' FOV is considerably long. Coupled with the fact that the attackers are always within the defenders' FOV due to the nature of pursuit guidance, the result is that the integration of the low hit rate values over a long period of time reduces the probability of the attackers surviving to a point where they are considered neutralized in the simulation.

This result is not observed when PN guidance is used because the attackers will go out of the defenders' FOV when the speed of the defenders gets too low. At higher speed ratios, this effect will not be significant as the region of very low hit rate values will be traversed in a reasonably short period of time that it does not contribute much to lowering the survival rate of the attackers.

We have to look into altering the modeling of the hit rate function of the defenders to limit the range at which the function can have a value. This will probably give more reasonable and more realistic results. This can be done in further studies on this subject.

IV. CONCLUSION AND RECOMMENDED FUTURE WORK

A. CONCLUSION

In this thesis, we have developed a simulation that allowed us to simulate various distributed strategies that can be used in the defense of a HVU against a swarm attack. A cost function is formulated to calculate probability that the HVU survives the swarm attack. This gives us a measure of effectiveness of each strategy used.

The results have given us great insights into how some of the defender parameters used can influence the survival rate of the HVU. Pursuit guidance, normally not very effective in modern missile applications, has shown to be a rather effective strategy, and seemingly even better than the preferred PN guidance law.

Having larger number of defenders improves the effectiveness of the defensive force, but beyond a certain number of defenders, the utility of the defenders will drop. The optimal ratio of defenders to attackers is shown to be 1 to 4 for the particular scenario that we looked at.

The simulations also showed that having high defender speeds might hinder, not help, the defense operation. A moderate speed that allows sufficient time on target, but yet able to intercept attackers at a reasonable distance from the HVU should be chosen.

The results where pursuit guidance at low speeds perform surprisingly well reveals that the hit rate function used in the simulations still needs some more considerations to make the analysis more accurate.

B. RECOMMENDED FUTURE WORK

1. Other Guidance Methods or Strategies

The simulation developed in this thesis provides a framework in which different guidance methods or defensive strategies can be evaluated for their effectiveness. For example, Shin, 2011[6] describes a Earliest Intercept Geometry Guidance Law (EIGGL) that can be used for target allocation to minimize the distance that the attacker can cover, and hence maximizes the distance of the attackers to the HVU.

An impact angle control guidance law is also described in the same paper. This guidance law optimizes the impact angle between the defender and the attacker during the time of intercept, effectively setting them up in a head-on collision course, which increases the chance of interception. These laws can be evaluated for their effectiveness in the swarm attack scenario that we have set up.

2. Different Engagement Scenarios

In this thesis, we have only looked at a particular engagement scenario where a HVU is moving in a straight line at a relatively slow speed, with attackers moving aggressively towards it from one direction. A HVU can come under a swarm attack in many other possible situations. For example, attackers can come from multiple directions, and may maneuver to avoid the defenders that are deployed against them. The HVU can be stationary, like an oilrig or even an on-shore facility. All these scenarios can be modeled and simulated within this framework, and different defensive strategies can be evaluated to find out their effectiveness in such scenarios.

3. Intent Recognition

At the initial stage of the research for this thesis, a real time graphical simulation with intent recognition algorithm was being developed in the University of Nevada, Reno (UNR). The intention was to integrate this simulation with the simulation developed here to simulate the engagement in real time and display the engagement in 3D graphics. More importantly, the intent recognition algorithm would allow us to study the reaction of the defensive force when the hostile targets are hidden among neutral ones. Daniel Bigelow from UNR, came down to NPS to install the simulation for the integration. Unfortunately, even though the simulation was successfully installed, the simulation was unstable due to the different hardware used here in NPS. Due to time constraints, this part of the thesis had to be abandoned, but future work can be done to further explore the possibilities of integrating the simulations.

4. Hit Rate Function Modeling

As discussed in the results section, the hit rate functions used currently may give us MOE that are misleading due to the small finite values of the hit rate function at large ranges. Further work is needed to come up with a hit rate function that has a finite range value and describes range effectiveness of real life weapons more accurately.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MATLAB® SIMULATION CODE

This appendix contains the list of script files and function codes that were used in the simulation.

A. SIMULATION RUN SCRIPT FILES

```
%----------------------------------------------------------------------
% File: Evaluate_Scenario_Performance.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Sets up the parameters for cost evaluation algorithm
% - Runs trajectory generation scripts
% - Computes cost and plots attacker and defender trajectories
% Inputs: Discretization time and space, method of evaluation, number
%           of attackers, number of defenders
% Outputs: Objective value
%----------------------------------------------------------------------
function ObjValue = Evaluate_Scenario_Performance(Discretization,
                                          Methods,
                                          NumberofAttackers,
                                          NumberofDefenders)

global CONSTANTS OFFLINE_TRAJECTORIES
       PDF_VALUES MESHED_PDF_VALUES
       DISCRETIZATION_VALUES MESHED_DISCRETIZATION_VALUES
       DIFFERENTIATION_MATRICES
       INTEGRATION_WEIGHTS MESHED_INTEGRATION_WEIGHTS;

global p0 N % To interface with Kaminer code without editing

warning('off', 'all')
% This suppresses the warning that future versions of MATLAB® will not
% support evaluating scripts with feval, which is what I'm using to
% evaluate arbitrary versions of Ding's heuristic

addpath('./performance_related_functions', '-begin');
addpath('./performance_related_functions/Parameterized_Control_Kernel')
addpath('./performance_related_functions/hit_rate_functions')
addpath('./performance_related_functions/cline')
addpath('./code', '-begin') %Ding's code

Heuristic = 'PlotOneManyBadGuysV11';

% PDF choices
CONSTANTS.ParameterSpace.PFD_Choices
                              ={'Independent','Uniform','Uniform'};

%---Simulation Time Range---%
CONSTANTS.Time.T0=0;    %start time
```

```
%-----Parameter Ranges------%
%----------------------------------------------------%
%   W0 and WF should be 1xParameterSpace.Dimension   %
% arrays with the starting points and ending points  %
% respectively of each parameter's domain.           %
% When attackers have identical parameter ranges,    %
% the shortcut of listing just the number of         %
% parameters per attacker can be taken.              %
%----------------------------------------------------%
CONSTANTS.ParameterSpace.W0=[8000,0];     %start of each parameter
CONSTANTS.ParameterSpace.WF=[8100,500];   %end of each parameter

%---------------------------------------------------------------------
% The following variables are not meant to be modified.
% They are set in the function call and just relabeled here to match
% with previous code.
%---------------------------------------------------------------------
N = Discretization(1);
CONSTANTS.N = N; % This redundant declaration is here, because the
% heuristics and Kaminer code need N as a global variable to be run
% with minimal editing (to be run without rewriting them as function
% calls), but the control code and many subroutines use CONSTANTS.N

% Number of probabilistic parameters per attacker. In this case,
% starting position (in two dimensions).
CONSTANTS.ParameterSpace.Dimension = 2;

% Number of attackers
% This is part of the substruct of attacker specific constants, for
% historical reasons.
CONSTANTS.ATTACKERS.Na = NumberofAttackers;
% Number of searchers
CONSTANTS.Ns = NumberofDefenders;
%---------------------------------------------------------------------

%---------------------------------------------------------------------
% Set the constants for the hit rate functions and the filenames
%---------------------------------------------------------------------
Calibrate_Hit_Rate_Functions
%---------------------------------------------------------------------

%---------------------------------------------------------------------
% Create DIFFERENTIATION_MATRICES, INTEGRATION_WEIGHTS,
% MESHED_INTEGRATION_WEIGHTS, DISCRETIZATION VALUES, and
% MESHED_DISCRETIZATION_VALUES.
%---------------------------------------------------------------------
disp('%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%');
disp(Heuristic);
disp('--------------------------------------------------------');
disp('Discretization:');
disp(Discretization);
Calculate_Methods(Discretization, Methods);
disp('--------------------------------------------------------');
%---------------------------------------------------------------------
```

```
 [x_a_i, x_d_a_i, x_0_a_i] =
                  RunDingsSimulation(Heuristic,NumberofDefenders,
                                    NumberofAttackers);
%---------------------------------------------------------------------

for s = 1:NumberofAttackers
    X =  interp1(x_a_i{s}(:,3),[x_a_i{s}(:,1),x_a_i{s}(:,2)],
                        DISCRETIZATION_VALUES{1},'spline');
    x_a_i{s}(:,3) = DISCRETIZATION_VALUES{1}(:);
    x_a_i{s}(:,1) = X(:,1);
    x_a_i{s}(:,2) = X(:,2);
end
for s = 1:NumberofDefenders
    X = interp1(x_d_a_i{s}(:,3),[x_d_a_i{s}(:,1),x_d_a_i{s}(:,2)],
                              DISCRETIZATION_VALUES{1},'spline');
    x_d_a_i{s}(:,3) = DISCRETIZATION_VALUES{1}(:);
    x_d_a_i{s}(:,1) = X(:,1);
    x_d_a_i{s}(:,2) = X(:,2);
end
X = interp1(x_0_a_i(:,3),[x_0_a_i(:,1),x_0_a_i(:,2)],
                  DISCRETIZATION_VALUES{1},'spline');
x_0_a_i(:,3) = DISCRETIZATION_VALUES{1}(:);
x_0_a_i(:,1) = X(:,1);
x_0_a_i(:,2) = X(:,2);
%---------------------------------------------------------------------

%---------------------------------------------------------------------
% Plot various desired plots.
%---------------------------------------------------------------------
Result_Plots;
%---------------------------------------------------------------------

%---------------------------------------------------------------------
% Calculate the probability function just for the a_i trajectories
%---------------------------------------------------------------------
[p, q] = Probability_Function(x_a_i,x_d_a_i,x_0_a_i);
% Probability that the HVU does not survive at time T.
ObjValue = 1-p(end);
%---------------------------------------------------------------------
```

```
%-----------------------------------------------------------------------
% File: Run_Scenarios.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Runs script to load 50 pre-generated attackers’ position
% - Runs Evaluate_Scenario_Performance.m with appropriate parameters.
% - Logs evaluated performance for the 50 runs
% Inputs: None
% Outputs: Array of objective values
%-----------------------------------------------------------------------
global p0

ObjValues = zeros(50,1);

for scn = 1:50
    p0 = load(['Scenario' int2str(scn) '.dat']);
    ObjValues(scn) = Evaluate_Scenario_Performance([500,1,1],[0,0,0],
                                                            40,5);
end
```

B. ATTACKER & DEFENDER TRAJECTORY GENERATION CODE

```
%---------------------------------------------------------------------
% File: RunMoreThanOneBadGuy.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Generates optimized attacker trajectory based on given
%                limitations
% Inputs: None
% Outputs: Time for optimized trajectory
%---------------------------------------------------------------------
global vShip psiShip v0 vf vmin vmax psidotmax p0 a1 lambda0 lambdaf tf
            p0ship BadGuyTotal N DefenderTotal
global p_swarm Defender p_ship Dindex tf

% Initial conditions for the ship
vShip = 5;        %m/sec
psiShip = pi/2;   %rad
p0ship = [0;0];

% Initial conditions for the bad guys
% Initial and final velocities of the bad guys in m/sec
v0 = 23; vf = 23;

% Limits for the bad guys
% Max and min speeds of the bad guy
vmin = 1; vmax = 23;
% Turn rate limit for the bad guy
psidotmax = 0.1;

% Initial guess on the hit time
tf = norm(p0(:,1) - p0ship)/vmax;
for i = 2:BadGuyTotal
    tf = min(tf,norm(p0(:,i) - p0ship)/vmax);
end

options = optimset('TolFun',.1,'maxiter',100,'MaxFunEvals',100,
                        'Display', 'on');
[x,fval,exitflag,output] =
                        fminsearch(@MoreThanOneBadGuyCost,tf,options);

% This ends the initialization of BadGuy part
tf = x(1);
```

```
%----------------------------------------------------------------------
% File: MoreThanOneBadGuy.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Computes cost for a single attacker
% Inputs: Number of attackers, final time
% Outputs: Cost for single attacker trajectory
%----------------------------------------------------------------------
function  J = MoreThanOneBadGuy(BadGuyNumber,tf0)

global a1 vShip psiShip v0 vf vmin vmax psidotmax p0 p0ship N

% Time step
dt = tf0/N;

% Extract optimization parameters
x0 = p0(:,BadGuyNumber);
psi0 = atan2(x0(2)-p0ship(2),x0(1)-p0ship(1))-pi;

% Initialize
xf = p0ship + [vShip*cos(psiShip);
vShip*sin(psiShip)]*(tf0-(BadGuyNumber - 1)*dt)';
xp0 = [cos(psi0); sin(psi0)];

perror = [cos(psiShip) sin(psiShip);-sin(psiShip) cos(psiShip)]
            *(x0 - p0ship);

if  perror(2) >= 0
    psif = psiShip - pi/2;
else
    psif = psiShip + pi/2;
end

xpf = [cos(psif); sin(psif)];
xpp0 = zeros(2,1); xppf = zeros(2,1);
xppp0 = zeros(2,1); xpppf = zeros(2,1);

% Define speed profile
lambda0 = v0/norm(xp0);
lambdaf = vf/norm(xpf);

% Evaluate UAV path coefficients
if (abs(lambda0 - lambdaf) < 1e-6)
    tauf = lambda0*tf0;
else
    tauf = (lambdaf - lambda0)*tf0/log(lambdaf/lambda0);
end
% The 7th order coefficients below are obtained using the same steps as
% in Ghabcheloo paper
% A = [1 0 0 0 0 0 0 0;
%      0 1 0 0 0 0 0 0;
%      0 0 2 0 0 0 0 0;
%      0 0 0 6 0 0 0 0;
%      1 tauf tauf^2 tauf^3 tauf^4 tauf^5 tauf^6 tauf^7;
%      0 1 2*tauf 3*tauf^2 4*tauf^3 5*tauf^4 6*tauf^5 7*tauf^6;
%      0 0 2 6*tauf 12*tauf^2 20*tauf^3 30*tauf^4 42*tauf^5;
%      0 0 0 6 24*tauf 60*tauf^2 120*tauf^3 210*tauf^4];
%
% b = [x0 xp0 xpp0 xppp0 xf xpf xppf xpppf]';
%
% a1 = inv(A)*b;

a1 = [x0 ...
      xp0 ...
      xpp0/2 ...
      xppp0/6 ...
      (35*xf)/tauf^4 - (35*x0)/tauf^4 - (20*xp0)/tauf^3 –
            (15*xpf)/tauf^3 – (5*xpp0)/tauf^2 + (5*xppf)/(2*tauf^2) –
            (2*xppp0)/(3*tauf) - xpppf/(6*tauf) ...
      (84*x0)/tauf^5 - (84*xf)/tauf^5 + (45*xp0)/tauf^4 +
            (39*xpf)/tauf^4 + (10*xpp0)/tauf^3 - (7*xppf)/tauf^3 +
            xppp0/tauf^2 + xpppf/(2*tauf^2) ...
      (70*xf)/tauf^6 - (70*x0)/tauf^6 - (36*xp0)/tauf^5 –
            (34*xpf)/tauf^5 - (15*xpp0)/(2*tauf^4) +
            (13*xppf)/(2*tauf^4) - (2*xppp0)/(3*tauf^3) –
            xpppf/(2*tauf^3) ...
      (20*x0)/tauf^7 - (20*xf)/tauf^7 + (10*xp0)/tauf^6 +
            (10*xpf)/tauf^6 + (2*xpp0)/tauf^5 - (2*xppf)/tauf^5 +
            xppp0/(6*tauf^4) + xpppf/(6*tauf^4)];

v_max = max(v0,vf); v_min = min(v0,vf); psidot_max = 0;

for n = 0:N

    t = n*dt;
    if (abs(lambda0 - lambdaf) < 1e-6)
        tau = t/tf0*tauf;
    else
        tau = tauf*((lambdaf/lambda0)^(t/tf0) - 1)*lambda0/(lambdaf-
                                                            lambda0);
    end

    % Compute UAV path and its derivatives
    p = zeros(2,1);
    for i = 1:8
        p = p + a1(:,i)*tau^(i-1);
    end
    p_p = zeros(2,1);
    for i = 2:8
        p_p = p_p + (i-1)*a1(:,i)*tau^(i-2);
    end
    p_pp = zeros(2,1);
    for i = 3:8
        p_pp = p_pp + (i-1)*(i-2)*a1(:,i)*tau^(i-3);
    end

    % Compute speed
    lambda = lambda0 + (lambdaf-lambda0)*tau/tauf;
    v = lambda*norm(p_p);
    % Compute signed curvature in 2D
    curv = (p_pp(2)*p_p(1) - p_pp(1)*p_p(2))/norm(p_p)^3;
    psidot_max = max(psidot_max,v*curv);
    v_max = max(v,v_max);
    v_min = min(v,v_max);

end

J = tf0 + 10*(psidot_max - psidotmax)^2/psidotmax^2 + (v_min –
            vmin)^2/vmin^2 + 1e6*(v_max - vmax)^2/vmax^2;

return
```

```
%-----------------------------------------------------------------------
% File: MoreThanOneBadGuyCost.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Computes total cost for all attackers
% Inputs: Final time
% Outputs: Total cost for all attackers
%-----------------------------------------------------------------------
function  J = MoreThanOneBadGuyCost(tf0)

global BadGuyTotal

J = tf0;

for i = 1:1:BadGuyTotal
    J = J + MoreThanOneBadGuy(i,tf0);
end

return
```

```
%-----------------------------------------------------------------------
% File: MoreThanOneBadGuyData.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Calculate attacker position and velocity
% Inputs: Number of attackers, final time, current simulation time
% Outputs: Position, speed, turn rate, velocity of attacker
%-----------------------------------------------------------------------
function  [p,v,psidot,vel] = MoreThanOneBadGuyData(BadGuyNumber,tf0,t)

global vShip psiShip v0 vf p0 p0ship N % vmin vmax psidotmax

% Time step
dt = tf0/N;

% Extract optimization parameters
x0 = p0(:,BadGuyNumber);
psi0 = atan2(x0(2)-p0ship(2),x0(1)-p0ship(1))-pi;

% Initialize
xf = p0ship + [vShip*cos(psiShip); vShip*sin(psiShip)]*(tf0-
                                    (BadGuyNumber - 1)*dt)';
xp0 = [cos(psi0); sin(psi0)];

perror = [cos(psiShip) sin(psiShip);-sin(psiShip) cos(psiShip)]*(x0-
                                                            p0ship);

if  perror(2) >= 0
    psif = psiShip - pi/2;
else
    psif = psiShip + pi/2;
end

xpf = [cos(psif); sin(psif)];
xpp0 = zeros(2,1); xppf = zeros(2,1);
xppp0 = zeros(2,1); xpppf = zeros(2,1);

% Define speed profile
lambda0 = v0/norm(xp0);
lambdaf = vf/norm(xpf);

% Evaluate UAV path coefficients
if (abs(lambda0 - lambdaf) < 1e-6)
    tauf = lambda0*tf0;
else
    tauf = (lambdaf - lambda0)*tf0/log(lambdaf/lambda0);
end
% The 7th order coefficients below are obtained using the same steps as
% in Ghabcheloo paper
% A = [1 0 0 0 0 0 0 0;
%      0 1 0 0 0 0 0 0;
%      0 0 2 0 0 0 0 0;
%      0 0 0 6 0 0 0 0;
%      1 tauf tauf^2 tauf^3 tauf^4 tauf^5 tauf^6 tauf^7;
%      0 1 2*tauf 3*tauf^2 4*tauf^3 5*tauf^4 6*tauf^5 7*tauf^6;
%      0 0 2 6*tauf 12*tauf^2 20*tauf^3 30*tauf^4 42*tauf^5;
%      0 0 0 6 24*tauf 60*tauf^2 120*tauf^3 210*tauf^4];
%
% b = [x0 xp0 xpp0 xppp0 xf xpf xppf xpppf]';
%
% a1 = inv(A)*b;

a1 = [x0 ...
      xp0 ...
      xpp0/2 ...
      xppp0/6 ...
      (35*xf)/tauf^4 - (35*x0)/tauf^4 - (20*xp0)/tauf^3 –
            (15*xpf)/tauf^3 – (5*xpp0)/tauf^2 + (5*xppf)/(2*tauf^2) –
            (2*xppp0)/(3*tauf) - xpppf/(6*tauf) ...
      (84*x0)/tauf^5 - (84*xf)/tauf^5 + (45*xp0)/tauf^4 +
            (39*xpf)/tauf^4 + (10*xpp0)/tauf^3 - (7*xppf)/tauf^3 +
            xppp0/tauf^2 + xpppf/(2*tauf^2) ...
      (70*xf)/tauf^6 - (70*x0)/tauf^6 - (36*xp0)/tauf^5 –
            (34*xpf)/tauf^5 - (15*xpp0)/(2*tauf^4) +
            (13*xppf)/(2*tauf^4) - (2*xppp0)/(3*tauf^3) –
            xpppf/(2*tauf^3) ...
      (20*x0)/tauf^7 - (20*xf)/tauf^7 + (10*xp0)/tauf^6 +
            (10*xpf)/tauf^6 + (2*xpp0)/tauf^5 - (2*xppf)/tauf^5 +
            xppp0/(6*tauf^4) + xpppf/(6*tauf^4)];

if (abs(lambda0 - lambdaf) < 1e-6)
    tau = t/tf0*tauf;
else
    tau = tauf*((lambdaf/lambda0)^(t/tf0) - 1)*lambda0/(lambdaf-
lambda0);
end

% Compute UAV path and its derivatives
p = zeros(2,1);
for i = 1:8
    p = p + a1(:,i)*tau^(i-1);
end
p_p = zeros(2,1);
for i = 2:8
    p_p = p_p + (i-1)*a1(:,i)*tau^(i-2);
end
p_pp = zeros(2,1);
for i = 3:8
    p_pp = p_pp + (i-1)*(i-2)*a1(:,i)*tau^(i-3);
end

% Compute speed
lambda = lambda0 + (lambdaf-lambda0)*tau/tauf;
v = lambda*norm(p_p);
vel = lambda*p_p;

% Compute signed curvature in 2D
curv = (p_pp(2)*p_p(1) - p_pp(1)*p_p(2))/norm(p_p)^3; % THIS IS IN M
psidot = v*curv;
end
```

```
%-----------------------------------------------------------------------
% File: PlotOneManyBadGuysV11.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - Each defender calculates a weighted centroid and effective velocity
% - for the attackers in its FOV.
% - A turn rate command is generated using PN guidance using the
% - calculated centroid and velocity.
% - This version divides up the attackers and assigns them to each
% - defender. The defenders will stop once they lose sight of attackers
% - assigned to them.
% - This version also includes a neutralizing algorithm, in which an
% - attacker is neutralized when it stays within the FOV and attacking
% - range of a defender for a set amount of time.
% Inputs: Initial ship and swarm parameters
% Outputs: Defender trajectory
%-----------------------------------------------------------------------
global vShip psiShip v0 vf N p0 DefenderTotal
global p_swarm Defender p_ship Dindex tf

% PlotOneManyBadGuysV11 - Guidance Version 11

dt = tf/N; % Time Step
p_ship = zeros(N+1,3);
for i = 0:N
    t = i*dt;
    p_ship(i+1,:) = [t p0ship' + [vShip*cos(psiShip)
vShip*sin(psiShip)]*t];
end

% Assigning attackers to defenders
% This algorithm assumes attackers >= defenders
% Attacker groups are assigned to defenders based on their position
Assign=zeros(DefenderTotal,2);
Assign(DefenderTotal,1)=1;
Assign(DefenderTotal,2)=floor(BadGuyTotal/DefenderTotal);
for i=DefenderTotal-1:-1:1
    Assign(i,1)=Assign(i+1,2)+1;
    Assign(i,2)=Assign(i+1,2)+floor(BadGuyTotal/DefenderTotal);
end
for i=1:mod(BadGuyTotal,DefenderTotal)
    Assign(1:i,2)=Assign(1:i,2)+1;
end
for i=1:mod(BadGuyTotal,DefenderTotal)-1
    Assign(1:i,1)=Assign(1:i,1)+1;
end
% Centroid of initial position of attackers
APos = zeros(2,DefenderTotal);
for j=1:DefenderTotal
    for i = Assign(j,1):Assign(j,2)
        APos(:,j) = APos(:,j) + p0(:,i);
    end
    APos(:,j) = APos(:,j)/(Assign(j,2)-Assign(j,1)+1);
end
```

```
% Defenders parameters
MaxOmega = 1;       % Max turn rate
Dfov = 120;         % Field of view (deg)
Dv = 25;            % Speed (m/s)

% Initialize matrices
range_max = 0; v_max = max(v0,vf); v_min = min(v0,vf); psidot_max = 0;
p_test = zeros(N+1,2);
p_swarm = zeros(N+1,(BadGuyTotal*2)+1); v_swarm =
zeros(N+1,BadGuyTotal+1); psi_dot = zeros(N+1,BadGuyTotal+1);
pcombined = zeros(1,BadGuyTotal*2); vcombined = zeros(1,BadGuyTotal);
psidotcombined = zeros(1,BadGuyTotal);
Dpos = zeros(2,DefenderTotal); Dpsi = zeros(DefenderTotal,1);
Di = zeros(2,DefenderTotal);Defender = zeros(N+1, DefenderTotal*2+1);
VelD = zeros(2,DefenderTotal);
Range = zeros(DefenderTotal,N+1,BadGuyTotal+1); Dindex =
zeros(DefenderTotal,1);
Alog = zeros(N+1, DefenderTotal*2+1); HeadingLog = zeros(N+1,
DefenderTotal+1);
NeutAtt = zeros(BadGuyTotal, DefenderTotal + 1);
SimEnd = 0;

% Initial positions and heading for defenders
for i = 1:DefenderTotal
    Dpos(:,i) = p0ship + (i-1)*[0;25] + [10;0];
    % Velocity vector pointed towards centroid of attackers
    VelD(:,i) = Dv*(APos(:,i)-Dpos(:,i))/norm(APos(:,i)-Dpos(:,i));
    Dpsi(i) = pi/2 - atan2(VelD(2,i), VelD(1,i));
end

for i = 0:N
    t = i*dt;
    p_ship(i+1,:) = [t p0ship' + [vShip*cos(psiShip)
                                    vShip*sin(psiShip)]*t];

    WPos = zeros(2, DefenderTotal);
    WVel = zeros(2, DefenderTotal);
    W = zeros(DefenderTotal,1);

    for j = 1:BadGuyTotal
        [p,v,psidot,vel] = MoreThanOneBadGuyData(j,tf,t);
        pcombined(1,((2*j)-1):(2*j)) = p';
        vcombined(1,j) = v;
        psidotcombined(1,j) = psidot;
        % If attacker is already neutralized skip to next one
        if (NeutAtt(j,DefenderTotal+1) ~= 0)
            continue;
        end

        for k = 1:DefenderTotal
            % Attacker not assigned to this defender, skip to next
                  defender
            if ((j<Assign(k,1))||(j>Assign(k,2)))
                continue;
            end
            % Calculate range to attacker
            Li = norm(p-Dpos(:,k));
            Range(k,i+1,1) = t;
            Range(k,i+1,j+1) = Li;

            Ai = (p-Dpos(:,k))/Li;  % Unit LOS vector to attacker
            dVel = VelD(:,k) - vel;
            Vc = Ai'*dVel; % Closing velocity

            % Calculate LOS angle
            temp = atan2(Ai(2),Ai(1));
            LOS = (temp-(pi/2-Dpsi(k)))*180/pi;
            while (abs(LOS) > 180)
                LOS = -1*sign(LOS)*(360 - abs(LOS));
            end
            % Sum weighted position and velocity if within FOV
            if (abs(LOS) <= Dfov/2)
                if (Li < 150)
                    % Attacker within FOV and attacking range of
                    % defender, increase counter (one time step)
                    NeutAtt(j,k) = NeutAtt(j,k) + 1;
                    % If attacker stays within attacking range for X
                    % secs attacker considered neutralized
                    if (NeutAtt(j,k)*dt > 3)
                        NeutAtt(j,DefenderTotal+1) = i;
                        continue;
                    end
                else
                    NeutAtt(j,k) = 0; % Reset neutralize counter if
                                          attacker goes out of range
                end
                WPos(:,k) = WPos(:,k) + (Vc/Li)*p;
                WVel(:,k) = WVel(:,k) + (Vc/Li)*vel;
                W(k) = W(k) + (Vc/Li);
            else
                NeutAtt(j,k) = 0; % Reset neutralize counter if
                                    attacker goes out of FOV
            end
        end
    end

    % If all attackers neutralized, end simulation
    if ((prod(NeutAtt(:,DefenderTotal+1)) ~= 0) && (SimEnd == 0))
        SimEnd = i;
    end

    % Guidance algorithm
    for k = 1:DefenderTotal
        Defender(i+1, 1) = t;
        Defender(i+1, (2*k):(2*k+1)) = Dpos(:,k)';
        % Defender lost sight of attacker, stop guidance
        if (abs(W(k)) < 1e-15)
            continue;
        else % Attacker(s) in FOV
            % Calculate weighted centroid and velocity
```

```
            CPos = WPos(:,k)/W(k);
            CVel = WVel(:,k)/W(k);
        end
        Alog(i+1, 1) = t;
        Alog(i+1, (2*k):(2*k+1)) = CPos';
        Li = norm(CPos - Dpos(:,k));    % Calculate range to centroid
        Ai = (CPos-Dpos(:,k))/Li;       % Unit LOS vector to centroid
        dVel = VelD(:,k) - CVel;
        Vc = Ai'*dVel;                  % Closing velocity of centroid
        % LOS rate of centroid
        LOSrate = (dVel(1)*Ai(2)-dVel(2)*Ai(1))/Li;
        temp = atan2(Ai(2),Ai(1));
        LOS = (temp-(pi/2-Dpsi(k)));
        % Calculate turn rate using PN guidance
        omega = 6*LOSrate;
        % Limit turn rate by MaxOmega
        if (abs(omega) > MaxOmega)
            omega = sign(omega)*MaxOmega;
        end
        turn = -1*omega*dt;                         % Turn angle
        Dpsi(k) = Dpsi(k) + turn;                   % Update heading
        VelD(:,k) = Dv*[sin(Dpsi(k));cos(Dpsi(k))]; % Update velocity
        Dpos(:,k) = Dpos(:,k) + VelD(:,k)*dt;       % Update position

        HeadingLog(i+1,1) = t;
        HeadingLog(i+1, k+1) = pi/2-Dpsi(k);
    end

    v_max = max(v,v_max);
    v_min = min(v,v_max);

    p_swarm(i+1,:) = [t pcombined];
    v_swarm(i+1,:) = [t vcombined];
    psi_dot(i+1,:) = [t psidotcombined];

    % Update centroid of position of attackers
    APos = zeros(2,DefenderTotal);
    for j=1:DefenderTotal
        for k = Assign(j,1):Assign(j,2)
            APos(:,j) = APos(:,j) + pcombined(1,2*k-1:2*k)';
        end
        APos(:,j) = APos(:,j)/(Assign(j,2)-Assign(j,1)+1);
    end

    p_test(i+1,:) = p';
end
```

```
%-----------------------------------------------------------------------
% File: PlotOneManyBadGuysV12.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - The unit LOS vector to every attacker in the defenders' FOV is
% - combined to obtain the heading required to intercept the attackers.
% - The LOS vectors are weighted by the attackers' range and velocity.
% - This version divides up the attackers and assigns them to each
% - defender. The defenders will stop once they lose sight of attackers
% - assigned to them.
% - This version also includes a neutralizing algorithm, in which an
% - attacker is neutralized when it stays within the FOV and attacking
% - range of a defender for a set amount of time.
% Inputs: Initial ship and swarm parameters
% Outputs: Defender trajectory
%-----------------------------------------------------------------------
global vShip psiShip v0 vf N p0 DefenderTotal
global p_swarm Defender p_ship Dindex tf

dt = tf/N; % Time Step
p_ship = zeros(N+1,3);
for i = 0:N
    t = i*dt;
    p_ship(i+1,:) = [t p0ship' + [vShip*cos(psiShip)
vShip*sin(psiShip)]*t];
end

% Assigning attackers to defenders
% This algorithm assumes attackers >= defenders
% Attacker groups are assigned to defenders based on their position
Assign=zeros(DefenderTotal,2);
Assign(DefenderTotal,1)=1;
Assign(DefenderTotal,2)=floor(BadGuyTotal/DefenderTotal);
for i=DefenderTotal-1:-1:1
    Assign(i,1)=Assign(i+1,2)+1;
    Assign(i,2)=Assign(i+1,2)+floor(BadGuyTotal/DefenderTotal);
end
for i=1:mod(BadGuyTotal,DefenderTotal)
    Assign(1:i,2)=Assign(1:i,2)+1;
end
for i=1:mod(BadGuyTotal,DefenderTotal)-1
    Assign(1:i,1)=Assign(1:i,1)+1;
end

% Centroid of initial position of attackers
APos = zeros(2,DefenderTotal);
for j=1:DefenderTotal
    for i = Assign(j,1):Assign(j,2)
        APos(:,j) = APos(:,j) + p0(:,i);
    end
    APos(:,j) = APos(:,j)/(Assign(j,2)-Assign(j,1)+1);
end
```

```
% Defenders parameters
MaxOmega = 1;       % Max turn rate
Dfov = 120;          % Field of view (deg)
Dv = 25;            % Speed (m/s)

% Initialize matrices
range_max = 0; v_max = max(v0,vf); v_min = min(v0,vf); psidot_max = 0;
p_test = zeros(N+1,2);
p_swarm = zeros(N+1,(BadGuyTotal*2)+1); v_swarm =
zeros(N+1,BadGuyTotal+1); psi_dot = zeros(N+1,BadGuyTotal+1);
pcombined = zeros(1,BadGuyTotal*2); vcombined = zeros(1,BadGuyTotal);
psidotcombined = zeros(1,BadGuyTotal);
Dpos = zeros(2,DefenderTotal); Dpsi = zeros(DefenderTotal,1);
Di = zeros(2,DefenderTotal);Defender = zeros(N+1, DefenderTotal*2+1);
VelD = zeros(2,DefenderTotal);
Range = zeros(DefenderTotal,N+1,BadGuyTotal+1); Dindex =
zeros(DefenderTotal,1);
Alog = zeros(N+1, DefenderTotal*2+1); HeadingLog = zeros(N+1,
DefenderTotal+1);
NeutAtt = zeros(BadGuyTotal, DefenderTotal + 1);
SimEnd = 0;

% Initial positions and heading for defenders
for i = 1:DefenderTotal
    Dpos(:,i) = p0ship + (i-1)*[0;25] + [10;0];
    % Velocity vector pointed towards centroid of attackers
    VelD(:,i) = Dv*(APos(:,i)-Dpos(:,i))/norm(APos(:,i)-Dpos(:,i));
    Dpsi(i) = pi/2 - atan2(VelD(2,i), VelD(1,i));
end

for i = 0:N
    t = i*dt;

    Di = zeros(2, DefenderTotal);
    DEN = zeros(DefenderTotal,1);
    for j = 1:BadGuyTotal
        [p,v,psidot,vel] = MoreThanOneBadGuyData(j,tf,t);
        pcombined(1,((2*j)-1):(2*j)) = p';
        vcombined(1,j) = v;
        psidotcombined(1,j) = psidot;

        for k = 1:DefenderTotal
            % Attacker not assigned to this defender, skip to next
            % defender
            if ((j<Assign(k,1))||(j>Assign(k,2)))
                continue;
            end
            % Calculate range to attacker
            Li = norm(p-Dpos(:,k));
            Range(k,i+1,1) = t;
            Range(k,i+1,j+1) = Li;

            Ai = (p-Dpos(:,k))/Li;  % Unit LOS vector to attacker
            dVel = VelD(:,k) - vel;
            Vc = Ai'*dVel; % Closing velocity
            % Calculate LOS angle
            temp = atan2(Ai(2),Ai(1));
            LOS = (temp-(pi/2-Dpsi(k)))*180/pi;
            while (abs(LOS) > 180)
                LOS = -1*sign(LOS)*(360 - abs(LOS));
            end
            % Sum weighted LOS vector if within FOV
            if (abs(LOS) <= Dfov/2)
                if (Li < 150)
                    % Attacker within FOV and attacking range of
                    % defender, increase counter (one time step)
                    NeutAtt(j,k) = NeutAtt(j,k) + 1;
                    % If attacker stays within attacking range for X
                    % secs attacker considered neutralized
                    if (NeutAtt(j,k)*dt > 3)
                        NeutAtt(j,DefenderTotal+1) = i;
                        continue;
                    end
                else
                    NeutAtt(j,k) = 0; % Reset neutralize counter if
                                          attacker goes out of range
                end
                Di(:,k) = Di(:,k) + (Vc/Li)*Ai;
                DEN(k) = DEN(k) + (Vc/Li);
            end
        end
    end

    % If all attackers neutralized, end simulation
    if ((prod(NeutAtt(:,DefenderTotal+1)) ~= 0) && (SimEnd == 0))
        SimEnd = i;
    end

    % Guidance algorithm
    for k = 1:DefenderTotal
        Defender(i+1, 1) = t;
        Defender(i+1, (2*k):(2*k+1)) = Dpos(:,k)';
        if (abs(DEN(k)) < 1e-15)
            continue;
        else % Attacker(s) in FOV
            Di(:,k) = Di(:,k)/DEN(k);
            Di(:,k) = Di(:,k)/norm(Di(:,k));
        end
        turn = pi/2-atan2(Di(2,k), Di(1,k)) - Dpsi(k);
        if (abs(turn) > MaxOmega*dt)
            turn = sign(turn)*MaxOmega*dt;
            Dpsi(k) = Dpsi(k) + turn;
            Di(:,k) = [sin(Dpsi(k));cos(Dpsi(k))];
        else
            Dpsi(k) = pi/2-atan2(Di(2,k), Di(1,k));
        end
        Dpos(:,k) = Dpos(:,k) + Dv*Di(:,k)*dt;

        HeadingLog(i+1,1) = t;
```

```
            HeadingLog(i+1, k+1) = pi/2-Dpsi(k);
        end

        v_max = max(v,v_max);
        v_min = min(v,v_max);

        p_swarm(i+1,:) = [t pcombined];
        v_swarm(i+1,:) = [t vcombined];
        psi_dot(i+1,:) = [t psidotcombined];

        p_test(i+1,:) = p';
        % Update centroid of position of attackers
        APos = zeros(2,DefenderTotal);
        for j=1:DefenderTotal
            for k = Assign(j,1):Assign(j,2)
                APos(:,j) = APos(:,j) + pcombined(1,2*k-1:2*k)';
            end
            APos(:,j) = APos(:,j)/(Assign(j,2)-Assign(j,1)+1);
        end
    end
```

C. PERFORMANCE RELATED FUNCTIONS

```
%----------------------------------------------------------------------
% File: Probability_Function.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - Calculate the probability function trajectories x_a,x_d, x_0.
% - Returns a cell array for q which holds the probabilities q{l} for
% - each attacker's survival over time. Returns an array p, which hold
% - the probability at each time point for HVU survival. Note that the
% - final objective function is 1-p.
% Inputs:
% Outputs:
%----------------------------------------------------------------------
function [p, q] = Probability_Function(x_a, x_d, x_0)

global CONSTANTS OFFLINE_TRAJECTORIES ...
       PDF_VALUES MESHED_PDF_VALUES...
       DISCRETIZATION_VALUES MESHED_DISCRETIZATION_VALUES ...
       DIFFERENTIATION_MATRICES ...
       INTEGRATION_WEIGHTS MESHED_INTEGRATION_WEIGHTS;

N = CONSTANTS.N;

q = cell(1,CONSTANTS.ATTACKERS.Na);
for l = 1:CONSTANTS.ATTACKERS.Na
    q{l} = zeros(N, 1);
    z = zeros(N, 1);
    for i = 1:N
        z(i) = 0;
        attacker_positions = cell(1,CONSTANTS.ATTACKERS.Na);
        for s = 1:CONSTANTS.ATTACKERS.Na
            attacker_positions{1,s} = x_a{s}(i, 1:2);
            %all attacker positions influence the defender hit rate
        end
        point_to_evaluate = x_a{l}(i, 1:2);
        for k = 1:CONSTANTS.Ns
            defender_position = x_d{k}(i,1:2);
            z(i) = z(i)+...
                  feval(str2func(CONSTANTS.DEFENDER_HIT_RATE_FUNCTION),
                                          k,...
                                          point_to_evaluate,...
                                          defender_position,...
                                          defender_heading,...
                                          attacker_positions,...
                                          CONSTANTS.ATTACKERS.Na);
        end
    end
    for i = 1:N
        % This line of code assumes that the integration weights
        % still converge over the partial interval. This still
        % needs to be confirmed, but it's definitely true for
        % Euler's method.
        q{l}(i) = exp(-INTEGRATION_WEIGHTS{1}(1:i)'*z(1:i));
    end
end
p = zeros(N, 1);
z = zeros(N, 1);
for i=1:(N-1)
    for l = 1:CONSTANTS.ATTACKERS.Na
        point_to_evaluate = x_0(i,1:2);
        attacker_position = x_a{l}(i, 1:2);
        %only the position of the particular attacker influences the
        %attacker hit rate
        z(i) = z(i)+...
               q{l}(i)*...
               feval(str2func(CONSTANTS.ATTACKER_HIT_RATE_FUNCTION),...
                               l,...
                               point_to_evaluate,...
                               attacker_position);
    end
end
for i = 1:N
    % This line of code assumes that the integration weights
    % still converge over the partial interval. This still
    % needs to be confirmed, but it's definitely true for
    % Euler's method.
    p(i) = exp(-INTEGRATION_WEIGHTS{1}(1:i)'*z(1:i,1));
end
%----------------------------------------------------------------------
```

```
%----------------------------------------------------------------------
% File: RunDingsSimulation.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - Runs Ding's simulation using set Heuristic, NumberofDefenders, and
% - Number of attackers. p0, the array of starting positions of the
% - attackers, is also assumed to be passed to this file as a global
% - variable (it's been set to global so that Kaminer code can be run
% - without editing). Returns cell arrays x_a and x_d, and matrix x_0,
% - reformatted to be in the form used in the optimal control set up.
% - Note: Trajectories still need to be interpolated to match up time
% - points.
% Inputs: Heuristic function, number of attackers, number of defenders
% Outputs: x_a, x_d, x_0
%----------------------------------------------------------------------
function [x_a, x_d, x_0] =
                  RunDingsSimulation(Heuristic,NumberofDefenders,...
                                               NumberofAttackers)

global N DefenderTotal BadGuyTotal p0
global p_swarm Defender p_ship Dindex tf

DefenderTotal = NumberofDefenders;  % Total number of defenders
BadGuyTotal = NumberofAttackers;    % Total number of attackers

RunMoreThanOneBadGuy
% runs some of the Kaminer code with whatever value p0 is set to.
% Note that this version of the Kaminer code has a preset HVU
% trajectory baked into it.

feval(str2func(Heuristic));
% runs whichever algorithm is set as Heuristic. This is also necessary
% for running the rest of the Kaminer code.
% Trajectories are then put in the format being used in the control
% implementation

x_a=cell(1,NumberofAttackers); %cell array of attacker trajectories
for j = 1:NumberofAttackers
    x_a{1,j}=zeros(N, 3);
    x_a{1,j}(:,3)=p_swarm(1:N,1); %moves time to the third column
    x_a{1,j}(:,1)=p_swarm(1:N,2*j);
    x_a{1,j}(:,2)=p_swarm(1:N, 2*j+1);
end

x_d=cell(1,NumberofDefenders); %cell array of defender trajectories
for j = 1:NumberofDefenders
    x_d{1,j}=zeros(N, 3);
    x_d{1,j}(:,3)=Defender(1:N,1); %moves time to the third column
    x_d{1,j}(:,1)=Defender(1:N,2*j);
    x_d{1,j}(:,2)=Defender(1:N,2*j+1);
    x_d{1,j}(:,4)=HeadingLog(1:N,j+1); %puts heading in the fourth
column

end
```

```
x_0=zeros(N, 3); %HVU trajectory

x_0(:,3)=p_ship(1:N,1); %moves time to the third column
x_0(:,1)=p_ship(1:N,2);
x_0(:,2)=p_ship(1:N,3);



%-----------------------------------------------------------------------
% File: Calibrate_Hit_Rate_Functions.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - Calibrates hit rate functions to desired shape
% Inputs: None
% Outputs: None
%-----------------------------------------------------------------------
global CONSTANTS

%-----------Hit Rate Constants---------%


for i=1:CONSTANTS.Ns
    CONSTANTS.DEFENDER_HIT_RATE{i}.alpha_theta = 3;
    CONSTANTS.DEFENDER_HIT_RATE{i}.beta_theta = 3;
      %parameters for distribution used for angle effectiveness
    CONSTANTS.DEFENDER_HIT_RATE{i}.mu_r = 6;
    CONSTANTS.DEFENDER_HIT_RATE{i}.sigma_r = 0.7;
      %parameters for distribution for radial distance effectiveness
    CONSTANTS.DEFENDER_HIT_RATE{i}.max_angle =  pi/3;
      %FOV extends plus or minus this angle from heading
    CONSTANTS.DEFENDER_HIT_RATE{i}.rho = 1200;
      %radius within which attackers divide defender attention/rate
    CONSTANTS.DEFENDER_HIT_RATE{i}.sigma = .01;
      %tiny number for standard deviation of normal cdf that smooth rho
    x_star = exp(CONSTANTS.DEFENDER_HIT_RATE{i}.mu_r –
                        (CONSTANTS.DEFENDER_HIT_RATE{i}.sigma_r)^2);
    CONSTANTS.DEFENDER_HIT_RATE{i}.normalizing_constant_r =
                              1/lognpdf(x_star, ...
                              CONSTANTS.DEFENDER_HIT_RATE{i}.mu_r,...
                              CONSTANTS.DEFENDER_HIT_RATE{i}.sigma_r);
end

for i=1:CONSTANTS.ATTACKERS.Na
    CONSTANTS.ATTACKER_HIT_RATE{i}.alpha = 3;
    CONSTANTS.ATTACKER_HIT_RATE{i}.beta = 18;
    CONSTANTS.ATTACKER_HIT_RATE{i}.c1 = 1;
    %changes magnitude of rate function
    CONSTANTS.ATTACKER_HIT_RATE{i}.c2 =  1/800;
    %inversely changes radius of rate function
end

CONSTANTS.DEFENDER_HIT_RATE_FUNCTION =
                              'FOV_Defender_Hit_Rate_logn_withdivider';
CONSTANTS.ATTACKER_HIT_RATE_FUNCTION = 'Full_Beta_Attacker_Hit_Rate';
```

```
%----------------------------------------------------------------------
% File: Interpret_Results.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% Inputs: None
% Outputs: None
%----------------------------------------------------------------------
N_guess=500;
NumberofDefenders=5;
NumberofAttackers=40;

[x_a, x_d, x_0, N_real]=RunDingsSimulation(N_guess,...
                                           NumberofDefenders,...
                                           NumberofAttackers)

colors = ['g';'r';'k';'c';'m'];
plotters = ['xg';'xr';'xk';'xc';'xm'];
figure; hold on
for j = 1:NumberofAttackers
    plot(x_a{j}(:,1),x_a{j}(:,2));
end
plot(x_0(:,1),x_0(:,2),'m', 'LineWidth', 3,'DisplayName', 'HVU');
for k = 1:NumberofDefenders
    plot(x_d{k}(:,1),x_d{k}(:,2), colors(k), 'LineWidth',
3,'DisplayName', ['Defender ' int2str(k)]);
end
hold off
```

```
%-----------------------------------------------------------------------
% File: Plot_Hit_Rates.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Plot hit rate functions
% Inputs: None
% Outputs: None
%-----------------------------------------------------------------------
global CONSTANTS

time_index = 10;

x_a = x_a_i;
x_d = x_d_a_i;
x_0 = x_0_a_i;

figure
colormap('jet');
hold on
axis([-5,8000,0,3200,0,1])
view([-6 51])
grid on;

spatial_incr = 35; %fineness of spatial mesh

[X,Y] = meshgrid(-5:spatial_incr:8000,0:spatial_incr:3200);
[i_length, j_length]=size(X);

for s = 1:NumberofAttackers
    Z = 0*X;
    for i = 1:i_length
        for j = 1:j_length
            point_to_evaluate = [X(i,j),Y(i,j)];
            attacker_position = x_a{s}(time_index, 1:2);
            Z(i,j) =
                  feval(str2func(CONSTANTS.ATTACKER_HIT_RATE_FUNCTION),
                                    s,...
                                    point_to_evaluate,...
                                    attacker_position);
        end
    end
    mesh(X,Y,Z);
end

for s = 1:NumberofDefenders
    Z = 0*X;
    for i = 1:i_length
        for j = 1:j_length
            point_to_evaluate = [X(i,j),Y(i,j)];
            defender_position = x_d{s}(time_index, 1:2);
            defender_heading = x_d{s}(time_index,4);
            attacker_positions = cell(1,CONSTANTS.ATTACKERS.Na);
            for a = 1:CONSTANTS.ATTACKERS.Na
                attacker_positions{a} = x_a{a}(time_index, 1:2);
            end
            Z(i,j) =
                  feval(str2func(CONSTANTS.DEFENDER_HIT_RATE_FUNCTION),
                               s,....
                               point_to_evaluate,...
                               defender_position,...
                               defender_heading,...
                               attacker_positions,...
                               CONSTANTS.ATTACKERS.Na);
        end
    end
    mesh(X,Y,Z);
end

%---------------------------------------------------------------------
for s = 1:NumberofAttackers
        cline2D(x_a_i{s}(:,1),x_a_i{s}(:,2),q{s}(:),'Cool');
end
cline2D(x_0_a_i(:,1),x_0_a_i(:,2),p(:),'Autumn');

colormap('jet');
for s = 1:NumberofDefenders
    plot(x_d_a_i{s}(:,1), x_d_a_i{s}(:,2),'g', 'LineWidth',3);
end
%---------------------------------------------------------------------
```

```
%----------------------------------------------------------------------
% File: Result_Plots.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Plot all trajectories
% Inputs: None
% Outputs: None
%----------------------------------------------------------------------

global CONSTANTS OFFLINE_TRAJECTORIES ...
       PDF_VALUES MESHED_PDF_VALUES...
       DISCRETIZATION_VALUES MESHED_DISCRETIZATION_VALUES ...
       DIFFERENTIATION_MATRICES ...
       INTEGRATION_WEIGHTS MESHED_INTEGRATION_WEIGHTS;

NumberofAttackers = CONSTANTS.ATTACKERS.Na;
NumberofDefenders = CONSTANTS.Ns;

%---------------------------------------------------------------------
% Plot a_i trajectories
%---------------------------------------------------------------------
figure
view(2)
hold on
    for s = 1:NumberofAttackers
        cline2D(x_a_i{s}(:,1),x_a_i{s}(:,2),q{s}(:),'Cool');
    end
for s = 1:NumberofDefenders
    plot(x_d_a_i{s}(:,1), x_d_a_i{s}(:,2),'g');
end
cline2D(x_0_a_i(:,1),x_0_a_i(:,2),p(:),'Autumn');
%---------------------------------------------------------------------
figure
hold on
plot(x_0_a_i(:,3), p(:),'b')
for s = 1:NumberofAttackers
    plot(x_a_i{s}(:,3),q{s}(:),'r');
end
```

```
%----------------------------------------------------------------------
% File: cline2D.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - This function plots a 3D line (x,y,z) encoded with scalar color
% - data (c) using the specified colormap (default=jet);
% - SYNTAX: h=cline2D(x,y,z,c,colormap);
% - DBE 09/03/02
% Inputs:
% Outputs:
%----------------------------------------------------------------------
% Edited by CLW, 2012, to make 2D, and to make spline work when entries
% are identical
function h=cline2D(x,y,c,cmap);

if nargin==0  % Generate sample data...
  x=linspace(-10,10,101);
  y=2*x.^2+3;
  z=sin(0.1*pi*x);
  c=exp(z);
  w=z-min(z)+1;
  cmap='jet';
elseif nargin<3
  fprintf('Insufficient input arguments\n');
  return;
elseif nargin==3
  cmap='jet';
end

cmap=colormap(cmap);% Set colormap

% Generate range of color indices that map to cmap
yy=linspace(0,1,size(cmap,1));
cm = spline(yy,cmap',c);% Find interpolated colorvalue
cm(cm>1)=1;
cm(cm<0)=0;

% Lot line segment with appropriate color for each data pair...
for i=1:length(x)-1
      h(i)=line([x(i) x(i+1)],[y(i) y(i+1)],
                              'color',[cm(:,i)],'LineWidth',2);
end

return
```

```
%----------------------------------------------------------------------
% File: Full_Beta_Attacker_Hit_Rate.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Defines full beta attacker hit rate function
% Inputs: attacker_index, point_to_evaluate, attacker_position
% Outputs: rate
%----------------------------------------------------------------------
function rate =  Full_Beta_Attacker_Hit_Rate(attacker_index,...
                                             point_to_evaluate,...
                                             attacker_position)

global CONSTANTS

alpha = CONSTANTS.ATTACKER_HIT_RATE{attacker_index}.alpha;
beta = CONSTANTS.ATTACKER_HIT_RATE{attacker_index}.beta;
c1 = CONSTANTS.ATTACKER_HIT_RATE{attacker_index}.c1;
c2 = CONSTANTS.ATTACKER_HIT_RATE{attacker_index}.c2;

x_star = (alpha-1)/(alpha+beta-2); %point of function maximization
normalizing_constant = 1/betapdf(x_star, alpha, beta);
distance = c2*norm(attacker_position-point_to_evaluate);

if distance < 1
    rate = c1*normalizing_constant*betapdf(distance, alpha, beta);
else
    rate = 0;
end
```

```
%----------------------------------------------------------------------
% File: FOV_Defender_Hit_Rate_logn_withdivider
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Defines defender hit rate function
% Inputs: defender_index, point_to_evaluate, defender_position,
%           defender_heading, attacker_positions, NumberofAttackers
% Outputs: rate
%----------------------------------------------------------------------
function rate =  FOV_Defender_Hit_Rate_logn_withdivider
                                           (defender_index,...
                                            point_to_evaluate,...
                                            defender_position,...
                                            defender_heading,...
                                            attacker_positions,...
                                            NumberofAttackers)

global CONSTANTS

alpha_theta = CONSTANTS.DEFENDER_HIT_RATE{defender_index}.alpha_theta;
beta_theta = CONSTANTS.DEFENDER_HIT_RATE{defender_index}.beta_theta;
mu_r = CONSTANTS.DEFENDER_HIT_RATE{defender_index}.mu_r;
sigma_r = CONSTANTS.DEFENDER_HIT_RATE{defender_index}.sigma_r;
max_angle = CONSTANTS.DEFENDER_HIT_RATE{defender_index}.max_angle;
normalizing_constant_r =
CONSTANTS.DEFENDER_HIT_RATE{defender_index}.normalizing_constant_r;
rho = CONSTANTS.DEFENDER_HIT_RATE{defender_index}.rho;
sigma = CONSTANTS.DEFENDER_HIT_RATE{defender_index}.sigma;

Attention_Dividing_Sum = 1;
%We want the hit rate to stay constant for one attacker, but decrease
%proportionate to additional nearby attackers. This can be accomplished
%by the following shenanigans: If there is just one attacker the
%attention dividing sum is one. if more attackers, the nearest attacker
%is automatically counted as the 1. then each additional attacker is
%added on.

if NumberofAttackers>1
    %first need to figure out which attacker is nearest.
    distance = norm(attacker_positions{1,1}(:)-defender_position(:));
    index_of_min=1;
    for i=2:NumberofAttackers
        if norm(attacker_positions{1,i}(:)-
                                    defender_position(:))<distance;
            distance = norm(attacker_positions{1,i}(:)-
                                          defender_position(:));
            index_of_min = i;
        end
    end
    for i=1:NumberofAttackers
        if i~=index_of_min
            distance = norm(attacker_positions{1,i}(:)-
                                                defender_position(:));
            Attention_Dividing_Sum = Attention_Dividing_Sum +
                                    normcdf(rho-distance, 0, sigma);
        end
    end
end

distance = norm(point_to_evaluate-defender_position);

f_r = normalizing_constant_r*lognpdf(distance,mu_r, sigma_r);

angle_of_point = atan2(point_to_evaluate(2)-defender_position(2),...
              point_to_evaluate(1)-defender_position(1));

defender_heading = mod(defender_heading,2*pi); %mod so it's easier

% adjust heading to fit into atan2's format for polar coordinates
if defender_heading>pi
    defender_heading=defender_heading-2*pi;
end

%angle_of_point
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% There's a discontinuity at pi/-pi. The following cases deal
% with that.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if pi-max_angle<=defender_heading && defender_heading<=pi
    if -pi<=angle_of_point && angle_of_point<=-pi+max_angle
       angle_of_point = angle_of_point+2*pi;
    end
end
if -pi<=defender_heading && defender_heading<=-pi+max_angle
  if pi-max_angle<=angle_of_point && angle_of_point<=pi
    angle_of_point = angle_of_point-2*pi;
  end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

delta_theta = (abs(defender_heading-
angle_of_point)+max_angle)/(2*max_angle);
if (delta_theta>0 && delta_theta<1)
    x_star = (alpha_theta-1)/(alpha_theta+beta_theta-2); %point of
function maximization
    normalizing_constant = 1/betapdf(x_star, alpha_theta, beta_theta);
    f_theta = normalizing_constant*betapdf(delta_theta,alpha_theta,
beta_theta);
else
    f_theta=0;
end

rate = f_r*f_theta/Attention_Dividing_Sum;
```

```
%----------------------------------------------------------------------
% File: Calculate_Methods.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - This file is problem independent and should never have to be
% - edited, except to add more methods.
% - This function creates differentiation weights and integration
% - weights for a given method. It also creates the arrays for the
% - discretized values of the variables. It therefore accesses data
% - sent from the Problem File, by using the global CONSTANTS. The
% - weights it creates are problem specific; they do not have to be
% - renormalized for the variable intervals.
% Inputs: Discretization, Methods
% Outputs: None
%----------------------------------------------------------------------
% © 2012, CLAIRE WALTON. Some Rights Reserved.
%======================================================================
% Notes
% 'Discretization' should be an array with the values for how fine
% the discretization is for time and parameter space.
% E.g. in this case discretization [5,9,9] would run the simulation for
% 5 time steps and a 9x9 parameter space.
%
% 'Methods' should be an array with the same dimension as
% Discretization. Each entry is the method used for each discretization
% variable.
%           Method 0: Euler
%           Method 1: Pseudospectral with lgl points
%======================================================================
function Calculate_Methods(Discretization, Methods)

global CONSTANTS OFFLINE_TRAJECTORIES ...
       JOINT_PDF MESHED_JOINT_PDF...
       DISCRETIZATION_VALUES MESHED_DISCRETIZATION_VALUES ...
       DIFFERENTIATION_MATRICES ...
       INTEGRATION_WEIGHTS MESHED_INTEGRATION_WEIGHTS

%-----------------------------------
% Compute details for Time Domain
%-----------------------------------

    %-----------------------------------%
    % Note                              %
    %   Currently, the time domain is   %
    % being calculated with N+1, but    %
    % parameter space is using N.       %
    % This was inherited from previous  %
    % code and will probably be changed.%
    %-----------------------------------%

if Methods(1)==0
    disp('--------------------------------------------------------')
    disp('Methods:');
    disp('Time: Euler');
    [Dn, Time_Array, Weights]=euldiff(Discretization(1));
end

if Methods(1)==1
    disp('--------------------------------------------------------')
    disp('Methods:');
    disp('Time: Pseudospectral with Legendre Points');
    [Dn, Time_Array, Weights]=legdiff(Discretization(1));
% Get differentiation matrix, legendre points, and quadrature weights
end
    %-----------Transformation----------%
    % Have to transform these from the  %
    % interval [-1,1] to [T0,TF]. Equiv %
    % to u = (1/2)(TF-T0)t+(1/2)(TF+TO) %
    %-----------------------------------%
T0=CONSTANTS.Time.T0;
TF=CONSTANTS.Time.TF;
DISCRETIZATION_VALUES{1} = .5*(TF+T0)+.5*(TF-T0).*Time_Array;
    % cell array, DISCRETIZATION_VALUES{1} is the array of time points
DIFFERENTIATION_MATRICES{1} = (2/(TF-T0)).*Dn;
    % cell array, first element is Dn for time.
INTEGRATION_WEIGHTS{1} = .5*(TF-T0).*Weights;
    % cell array, first element is integration weights for time

%-----------------------------------
% Compute details for Parameter Space
%-----------------------------------
Size=CONSTANTS.ParameterSpace.Dimension+1;
    % i.e. length (Discretization)
for i=2:Size

    if Methods(i)==0
        str=['Parameter ',num2str(i-1), ': Euler'];
        disp(str);
        [Dn, Parameter_Array, Weights]=euldiff(Discretization(i));
    end

    if Methods(i)==1
        str=['Parameter ',num2str(i-1),
                        ': Pseudospectral with Legendre Points'];
        disp(str);
        [Dn, Parameter_Array, Weights]=legdiff(Discretization(i));
% Get differentiation matrix, legendre points, and quadrature weights
    end

    W0=CONSTANTS.ParameterSpace.W0(i-1);
    WF=CONSTANTS.ParameterSpace.WF(i-1);
    DISCRETIZATION_VALUES{i} = .5*(WF+W0)+.5*(WF-W0).*Parameter_Array;
% cell array, DISCRETIZATION_VALUES{1} is the array of time points

    DIFFERENTIATION_MATRICES{i} = (2/(WF-W0)).*Dn;
        % cell array, first element is Dn for time.

    INTEGRATION_WEIGHTS{i} = .5*(WF-W0).*Weights;
        % cell array, first element is integration weights for time
```

```
end
disp('--------------------------------------------------------')

%%%%%Create meshed values for multiple attackers%%%%

% there is no mesh for the time domain, but it's nice to keep the
% dimension numbers matching the unmeshed values
MESHED_DISCRETIZATION_VALUES{1}=[];
MESHED_INTEGRATION_WEIGHTS{1}=[];

% This creates the meshes for evaluating every permutation of
% discretization values. The objective function still needs to have a
% nested for loop iterating through the two meshes. Each i-th column
% of the cell is the value for the i-th attacker. Each row is a unique
% permutation of Na values. Equivalent meshes are created for the
% integration weights to keep track of which weights need to be used
% for each permutation. To integrate, take the product over all Na
% columns.

for i=2:Size
% first cell arrays are used to trick ndgrid into outputting the right
% dimension grid
    discretization_values=cell(1,CONSTANTS.ATTACKERS.Na);

    integration_weights=cell(1,CONSTANTS.ATTACKERS.Na);

    if CONSTANTS.ATTACKERS.Na>1
        [discretization_values{1,:}] =
                                    ndgrid(DISCRETIZATION_VALUES{i});
        [integration_weights{1,:}] = ndgrid(INTEGRATION_WEIGHTS{i});
    else
        discretization_values{1,:} = DISCRETIZATION_VALUES{i};
        integration_weights{1,:} = INTEGRATION_WEIGHTS{i};
    end
% then everything is reshaped to reduce notational confusion. While
% the meshgrid outputs Na Na-dimensional arrays, these can be indexed
% through and reread as Na one-dimensional vectors. This creates a
% matrix with the dimensions (parameter_length^Na, Na)
    MESHED_DISCRETIZATION_VALUES{i} =
                  zeros(length(discretization_values{1,1}(:)), ...
                                            CONSTANTS.ATTACKERS.Na);
    MESHED_INTEGRATION_WEIGHTS{i} =
                  zeros(length(discretization_values{1,1}(:)), ...
                                            CONSTANTS.ATTACKERS.Na);
   for s=1:CONSTANTS.ATTACKERS.Na
       MESHED_DISCRETIZATION_VALUES{i}(:,s) =
                                    discretization_values{1,s}(:);
       MESHED_INTEGRATION_WEIGHTS{i}(:,s) =
                                          integration_weights{1,s}(:);
   end
end
```

```
%----------------------------------------------------------------------
% File: euldiff.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - This function calculates the differentiation matrix Dn that is
% - obtained by using forward Euler's method on n equally spaced
% - points. These values are calculated over the interval [-1,1] and
% - need to be transformed to the interval of the problem. Note that
% - this does not return a square matrix. It returns a (N-1)xN matrix
% Inputs: N
% Outputs: Dn, x, w
%----------------------------------------------------------------------
% © 2012, CLAIRE WALTON. Some Rights Reserved.
%======================================================================
function [Dn,x,w]=euldiff(N);
h=2/(N-1);
x = (-1:h:1)';
w = ones(N,1).*h;
w(N,1)=0;
v1=ones(1,N-1);
v2=-1*ones(1, N);
temp=(diag(v1,1)+diag(v2))./h;
Dn = temp(1:N-1,:);
```

```
%----------------------------------------------------------------------
% File: legdiff.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - This function calculates the differentiation matrix Dn that is
% - obtained by differentiating the Lagrange Polynomials at the
% - Legendre-Gauss-Lobatto (LGL) points. It's  zero on the main
% - diagonal except at l=k=1, where Dn(1,1)= n(n+1)/4; and at l=k=n;
% - where Dn(n,n)=-n(n+1)/4. n= no of LGL points. For the other LGL
% - points l (~=)k, we have Dn(l,k)= Ln(xl)/Ln(xk)*(1/xl-xk).
% - This routine is part of DIDO Version 0.1
% - Written by Fariba Fahroo, edited by I. Michael Ross
% - Major subfunctions written by W. Gragg
% - Naval Postgraduate School, Monterey, CA 93943
% Inputs: n
% Outputs: Dn, x, w
%----------------------------------------------------------------------
function [Dn,x,w]=legdiff(n);

[x w]=lobatto(n);
x=sort(x);
% initialize Dn
Dn=zeros(n);
Dn(1,1)=-(n-1)*n/4;
Dn(n,n)=n*(n-1)/4;

% Calculate the legendre polynomials at xi.
p=0*eye(n);
for i=1:n; s=x(i); p(i,1)=1; p(i,2)=s;
for j=2:n-1; p(i,j+1)=((2*j-1)*s*p(i,j)-(j-1)*p(i,j-1))/j; end; end;

% Fill out the rest of matrix Dn.
for l=1:n; for k=1:n;
if l~=k,
Dn(l,k)=p(l,n)/(p(k,n)*(x(l)-x(k)));
end;
end;end;

%====================================
function [x,w] = lobatto(n,a,b)

% [x w] = lobatto(n) or [x w] = lobatto(n,alpha,beta):
%
% Computes abscissa and weights for the n-point Gauss-Jacobi-Lobatto
% quadrature formula using the method of Gene H. Golub, Some modified
% matrix eigenvalue problems, SIAM Rev. 15 (1973) 318-334.  Another
% early algorithm for this is by David Galant, An implementation of
% Christoffel's formula in the theory of orthogonal polynomials, Math.
% Comp. 25 (1971) 111-113.  All such algorithms should be "reviewed",
% in light of recent improvements in tqr and Cholesky LR algorithms.
% But, this algorithm "ain't bad".
% Copyright (c) 23 August 1997 by Bill Gragg.  All rights reserved.

% lobatto calls mxt, mxtj and tqr.
% begin lobatto

if nargin < 2
      a = 0;   b = 0;
end

m = 2^(a + b + 1)*beta(a+1,b+1);
us = a == b;

n = n - 1;        [a b] = mxtj(n,a,b);    T = mxt(a,b);
I = eye(n);       e = zeros(n,1);         e(n) = 1;
c = (T + I)\e;    c = c(n);               d = (T - I)\e;
d = d(n);         e = c - d;              c = (c + d)/e;
d = sqrt(2/e);    a(n+1) = c;             b(n) = d;
[x u] = tqr(a,b); u = u';                 w = m*u.^2;

% "Purify" formulas in the ultraspherical case.

if us
      x = (x - flipud(x))/2;   w = (w + flipud(w))/2;
end

% end lobatto
%-----------------------------------------------------------------
function T = mxt(a,b,c)
% T = mxt(a,b,c) or T = mxt(a,b):
%
% T = mxt(a,b,c) is the TRIDIAGONAL MATRIX with diagonal elements
% a(1:n), subdiagonal elements b(1:n-1) and superdiagonal elements
% c(1:n-1).
% T = mxt(a,b) is the HERMITIAN tridiagonal matrix with diagonal
% elements real(a(1:n)) and subdiagonal elements b(1:n-1).
% Copyright (c) 1 December 1990 by Bill Gragg.  All rights reserved.
% Revised 21 November 1992.

% mxt calls no extrinsic functions.

% begin mxt
if nargin < 3
      a = real(a);      c = b';
end

n = length(a);    b = b(1:n-1);
c = c(1:n-1);     z = zeros(n-1,1);

if n < 500
      B = diag(b);      B = [z' 0; B z];
      C = diag(c);      C = [z C; 0 z'];
      T = diag(a);      T = T + B + C;
else
      T = zeros(n);


      for k = 1:n-1
```

```
            T(k,k) = a(k);   T(k+1,k) = b(k);   T(k,k+1) = c(k);
      end

      T(n,n) = a(n);

end
% end mxt

%----------------------------------------------------------------------
function [a,b] = mxtj(n,alpha,beta)

% [a b] = mxtj(n,alpha,beta), [a b] = mxtj(n,alpha), [a b] = mxtj(n),
%     T = mxtj(n,alpha,beta),     T = mxtj(n,alpha) or   T = mxtj(n):
%
% mxtj(n,alpha,beta):  T = mxt(a,b) is the Jacobi matrix whose
% characteristic polynomial p is (a nonzero scalar multiple of) the nth
% JACOBI polynomial.
% The eigenvalues of T are the abscissas of the nth order Gauss-
% Christoffel quadrature formula for the weight function ((1 –
% t)^alpha)((1 + t)^beta) on the interval - 1 < t < 1.  The Gauss-
% Christoffel weights are m(0) times the squares of the first elements
% of the normalized eigenvectors of T, where m(0) = b(0)^2 = B(alpha +
% 1,beta + 1)2^(alpha + beta - 1) is the total mass.
% B is the beta function.  The weight function is positive and
% integrable if alpha + 1 > 0 and beta + 1 > 0.
%
% mxtj(n,alpha) takes beta = alpha.  p is the nth ULTRASPHERICAL
% polynomial, with weight function (1 - t^2)^alpha on the interval
% 1 < t < 1.  Special cases are the CHEBYSHEV polynomial of the FIRST
% KIND, with alpha = - 1/2, and of the SECOND KIND, with alpha = 1/2.
%
% mxtj(n) takes alpha = beta = 0.  p is the nth LEGENDRE polynomial,
% with weight function w(t) = 1 on the interval - 1 < t < 1.  The
% quadrature formula here is originally due to Gauss.  Christoffel
% generalized Gauss' formula to a wide class of weight functions.
% Because of this the Gauss-Christoffel weights are usually called
% Christoffel numbers.

% Copyright (c) 2 February 1991 by Bill Gragg.  All rights reserved.

% mxtj calls mxt.

% begin mxtj
if nargin < 2
      alpha = 0;
end;
if nargin < 3
      beta = alpha;
end

a = alpha;   b = beta;   c = a + b;   d = b - a;
s(1) = d/(c + 2);   t(1) = (a + 1)*(b + 1)/(c + 2)^2/(c + 3);

if n > 2
      d = c*d;
      n = (2:n)';   m = 2*n;   mm = m - 1;   mp = m + 1;
      s(n) = d./(c + m)./(c + (m - 2));
      t(n) = n.*(a + n).*(b + n).*(c + n)./(c + mm)./
                                                ((c + m).^2)./(c + mp);
end
a = s(:);   b = 2*sqrt(t(:));
if nargout < 2
      a = mxt(a,b);
end
% end mxtj

% Problems.
% 1. Relate T = mxt(a,b), with [a b] = mxtj(n,1/2), with the negative
%    second difference matrix S = mxt(c,d), with [c d] = mxs(n).
%----------------------------------------------------------------------
function [lam,U] = tqr(a,b,U)
%     [lam u] = tqr(a,b) or [lam U] = tqr(a,b,U):
%
%     [lam u] = tqr(a,b):
%
% The column lam contains the eigenvalues of the Hermitian tridiagonal
% matrix T = mxt(a,b) computed by one version of the (real symmetric)
% tqr algorithm with Wilkinson's shift.  The column u contains the
% first elements of the eigenvectors of T normalized to be nonnegative
% and such that the eigenvectors are unit vectors.  In practice this is
% an O(n^2) process.  If u is omitted only the eigenvalues are
% computed. The computed eigenvalues are real and are sorted to be
% nonincreasing.
%
%     [lam U] = tqr(a,b,U):
%
% This replaces the input U by UV with V a matrix of orthonormal eigen-
% vectors of T.  If the input U is I the output U is V.  If the input U
% is unitary with AU = UT then the output U is unitary with AU = UD and
% D = diag(lam).
%
% If the input U is e(1)' the output U is u'.  If the input U is
% [e(1)'; e(n)'] the output U is [u'; v'] with v the column of last
% elements of the normalized eigenvectors. If the subdiagonal elements
% of T are all nonzero then the elements of v alternate in sign, at
% least mathematically.

% Copyright (c) 2 February 1991 by Bill Gragg.  All rights reserved.
% Revised 15 July 1994.

% tqr calls sgn.
% begin tqr

% Ensure that T is Hermitian and shift b down one unit.
a = real(a);   n = length(a);   b = [0; b(:)];   b = b(1:n);

% Initialize U if required and execute a diagonal unitary similarity
% transformation to make T have nonnegative subdiagonal elements.

if nargout > 1
```

```
      if nargin < 3
            U = zeros(1,n);   U(1) = 1;
      end

      u = sgn(b);   u = cumprod(u);   U = U*diag(u);
end

b = abs(b);

% Scale the matrix up by a power of two to give nearly the widest
% possible exponent range.

scale = norm([a; b*sqrt(2)]);
scale = 2^(1024 - ceil(log2(scale)));
a = a*scale;
b = b*scale;

format compact                  % Temporary statements
maxscale = max(abs([a; b]));    % for display.

% "Do tqr".
for m = n:-1:1

% Compute the mth eigenvalue.
      for its = 0:10*n     % its is the iteration index.

% Split the matrix if possible.  This is also the termination test.
            for k = m:-1:1

                  if k > 1
                        tol = abs(a(k-1)) + abs(a(k));

                        if tol + b(k) == tol
                              b(k) = 0;  break
                        end
                  end
            end

            if k == m
                  break     % b(m) = 0.  a(m) is an eigenvalue.
            else

            if its == 10*n
                error('tqr iteration did not terminate in 10n steps!')
            end

% Compute Wilkinson's shift w as a perturbation of the
% Rayleigh shift r = a(m).  As the algorithm converges
% c = b(m) --> 0.
r = a(m);   c = b(m);   d = (r - a(m-1))/2;   s = abs(d);

if c < s
      s = c/s;   t = 1 + sqrt(1 + s*s);   t = c*s/t;   % t < c;
else
```

```
      s = s/c;   t = s + sqrt(1 + s*s);   t = c/t;     % t < c;
end

if d > 0
      w = r + t;
else
      w = r - t;
end

% Take a step of the tqr algorithm.  There are many ways to
% implement the inner loop.  We recently found the fastest
% known stable form in terms of flops.  The form given here
% is elegant.

c = 1;   s = 0;   p = w - a(k);   t = p;

for j = k:m-1
      % Compute the two by two reflector stably and update b(j).
      oldc = c;   oldt = t;   q = b(j+1);   u = abs(p);

      if q < u
            v = q/u;      r = sqrt(1 + v*v);   b(j) = u*r*s;
            u = sgn(p);   c = u/r;             s = v/r;
      else
            v = p/q;      r = sqrt(1 + v*v);   b(j) = q*r*s;
            u = 1;        c = v/r;             s = u/r;
      end

      % Update p, t, a(j), and U(:,j:j+1) if required.
      p = c*(w - a(j+1)) - s*q*oldc;   t = c*p;
      a(j) = a(j+1) + t - oldt;

      if nargout > 1
            i = j:j+1;   U(:,i) = U(:,i)*[-c s; s c];
      end
end

% Update b(m), a(m), and U(:,m) if required.
b(m) = abs(p)*s;   a(m) = w - t;   c = sgn(p);

if nargout > 1
      U(:,m) = - U(:,m)*c;
end

end

end

end

% Sort and prepare the output.
[a p] = sort(-a);   lam = - a/scale;

if nargout > 1
      U = U(:,p);   u = U(1,:)';
      if nargin < 3
            u = abs(u);   U = u';
      else
            u = sgn(u);   U = U*diag(u');
      end
end

% end tqr

%-------------------------------------------------------------------
function W = sgn(Z1,Z2)
% W = sgn(Z) or W = sgn(Z1,Z2):
%
% For z a complex number we define sgn z, the SIGNUM of z, as z/|z| if
% z ~= 0 and + 1 if z = 0.  Thus sgn z is the same as matlab's sign z
% except when z = 0.  We always have |sgn z| = 1, apart from rounding
% errors.
%
% With the first call W is the Schur (elementwise) sgn function of the
% complex matrix Z.  With the second call we have W = |Z1|.*sgn(Z2);

% Copyright (c) 19 January 1991 by Bill Gragg.  All rights reserved.
% Revised 29 May 1996.

% sgn calls no extrinsic functions.

% begin sgn
if nargin < 2
      W = sign(Z1);   W = W + (W == 0);
else
      W = sign(Z2);   W = W + (W == 0);   W = abs(Z1).*W;
end

% end sgn

% Total flops (scalar case, see csgn):  TBC

% Problem.
% 1. Compare this function experimentally with csgn. Compare with
% regard to both execution time and numerical stability. Is matlab
% computing sign correctly?
%=======================================================================
%----------------------------------------------------------------------
% File: Run_Optimization.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - Run_Optimization functions as a module that organizes an snopt call
% - for both searching and herding problem
% - Runs on general form agreed on for objective function
% Inputs: Problem_Info
% Outputs: x, F, xmul, Fmul, INFO, run_time
%----------------------------------------------------------------------
% © 2012, CLAIRE WALTON. Some Rights Reserved.
%======================================================================
% Notes
% 1. This file should be run after all offline trajectories/pdfs are
%    built
% 2. this file will compute sparsity patterns and run snopt
% 3. it will also split up the results into cost, control, and searcher
%    trajectories
% 4. run_time will measure snopt optimization, and will not include
%    parsing results
%
% Claire Walton, 01/19/12
%======================================================================
function [x,F,xmul,Fmul,INFO, run_time] =
                                    Run_Optimization(Problem_Info)

global CONSTANTS OFFLINE_TRAJECTORIES JOINT_PDF
DISCRETIZATION_VALUES DIFFERENTIATION_MATRICES INTEGRATION_WEIGHTS

Example1.spc = which('Example1.spc');
snspec (Example1.spc );

[Objective_lower, Objective_upper, Dynamics_lower, Dynamics_upper,....
 Variables_lower, Variables_upper] =
feval(str2func(Problem_Info.Optimization_Bounds));

xlow = Variables_lower;
xupp = Variables_upper;

Flow = [Objective_lower; Dynamics_lower];
Fupp = [Objective_upper; Dynamics_upper];
% F is the vector of functions made by concatenating the objective
% function and the dynamics constraints; these are its bounds.

nF = length(Flow);
% dimension of the function vector F
n = length(xlow);
% number of state variables

x = feval(str2func(Problem_Info.Initial_Guess));
%x = zeros(n,1); %this is for no initial guess
xstate = zeros(n,1);
% This is some weird thing that indicates something about values of x.
% Setting it equal to zero seems to be related to no initial guess
```

```
ObjAdd = 0;
% ObjAdd is a constant added to the objective function for printing
% purposes

ObjRow = 1;
% ObjAdd is the row in F containing the objective function. We will
% always use 1.

xmul   =     zeros(n,1);
% This has something to do with the vector of duals

Fstate =   zeros(nF,1);
Fmul   =   zeros(nF,1);
% I don't know what these are

Start = 1;
% The value of Start has something to do with the meaningfulness
% of xstate and Fstate. Start = 1 knows they're not meaningful.

% [iGfun,jGvar] = feval(str2func(Problem_Info.Gradient_Sparsity));
% [iAfun,jAvar,A] = feval(str2func(Problem_Info.Linear_Gradient));

disp('--------------------------------------------------------');
disp('  ... ... ... ... ... ... ... ... ... ... ... ... ...   ');
disp(' ... ... ... ... ... running snJac ... ... ... ... ...  ');
disp('  ... ... ... ... ... ... ... ... ... ... ... ... ...   ');

[A,iAfun,jAvar,iGfun,jGvar] =
snJac(Problem_Info.UserFun,x,xlow,xupp,nF);

snseti ('Derivative option', 0);
snseti ('Major Iteration limit', 1000);
snsetr ('Major feasibility tolerance', 10^(-4))
snsetr ('Minor feasibility tolerance', 10^(-5))
snsetr ('Major optimality tolerance', 10^(-4))
snsetr ('Minor optimality tolerance', 10^(-5))

run_time = cputime;

disp('--------------------------------------------------------');
disp('  ... ... ... ... ... ... ... ... ... ... ... ... ...   ');
disp(' ... ... ... ... ... running snopt ... ... ... ... ...  ');
disp('  ... ... ... ... ... ... ... ... ... ... ... ... ...   ');

[x,F,xmul,Fmul,INFO] = snoptcmex(Start, x, xlow, xupp, xmul, xstate,
                        Flow, Fupp, Fmul, Fstate, ObjAdd, ObjRow, A,...
                        iAfun, jAvar, iGfun, jGvar,...
                        Problem_Info.UserFun);

run_time = cputime - run_time;
disp('--------------------------------------------------------');
disp(' Run Time:');
disp(run_time);
disp('--------------------------------------------------------');
```

```
%----------------------------------------------------------------------
% File: Run_Problem.m
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description:
% - This file is problem independent and should never have to be
% - edited. It organizes the calls to three other functions:
% - 1. The function to create the differentiation matrices and
%      integration weights
% - 2. The function to build any attacker or hvu trajectories and pdfs
% - 3. The function the organizes the snopt call.
% - The purpose of the file is mostly to create the global variables.
% Inputs: Problem_Info, Discretization, Methods
% Outputs: Results, build_time, run_time, INFO
%----------------------------------------------------------------------
% © 2012, CLAIRE WALTON. Some Rights Reserved.
%======================================================================
function [Results, build_time, run_time, INFO] =
                  Run_Problem(Problem_Info,Discretization, Methods)

global CONSTANTS OFFLINE_TRAJECTORIES ...
       PDF_VALUES MESHED_PDF_VALUES...
       DISCRETIZATION_VALUES MESHED_DISCRETIZATION_VALUES ...
       DIFFERENTIATION_MATRICES ...
       INTEGRATION_WEIGHTS MESHED_INTEGRATION_WEIGHTS

build_time = cputime;

disp('%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%');
disp(Problem_Info.Name);
disp('--------------------------------------------------------');
disp('Discretization:');
disp(Discretization);

Calculate_Methods(Discretization, Methods);
%---------------------------------------------------------------------
% Creates DIFFERENTIATION_MATRICES, INTEGRATION_WEIGHTS, and
% DISCRETIZATION VALUES.
%
% DIFFERENTIATION_MATRICES and INTEGRATION_WEIGHTS are *cell arrays*
% with length given be length of discretization array. Each element of
% the cell array is the matrix/array given by calculated the
% differentiation matrix/integration array of the corresponding
% discretization variable.
% DISCRETIZATION_VALUES is also a cell array.
%---------------------------------------------------------------------

feval(str2func(Problem_Info.Build_Simulation));
%---------------------------------------------------------------------
% Contributes to the global variable TRAJECTORIES.
% TRAJECTORIES may include TRAJECTORIES.HVU and TRAJECTORIES.ATTACKERS,
% depending on the needs of the objective function userfun.
%---------------------------------------------------------------------

feval(str2func(Problem_Info.Build_PDF));
```

```
%---------------------------------------------------------------------
% Creates the joint PDF, as per the options chosen in Problem file.
% Put results in global cell array JOINT_PDF
%---------------------------------------------------------------------

build_time = cputime - build_time;
disp('--------------------------------------------------------');
disp(' Build Time:');
disp(build_time);
disp('--------------------------------------------------------');

[x,F,xmul,Fmul,INFO, run_time] = Run_Optimization(Problem_Info);
%---------------------------------------------------------------------
% Organizes the snopt call and calls it. Run_Optimization will access
% the global variables created here.
%---------------------------------------------------------------------

[Results] = ...
      feval(str2func(Problem_Info.Interpret_Results), x,F,xmul,Fmul);
```

```
%----------------------------------------------------------------------
% File: Uniform_PDF
% Compiler: MATLAB® v7.10.0.499 (R2010a)
% 64-bit (win64)
% Description: Forms uniform pdf
% Inputs: discretization_values, w0, wf
% Outputs: uniform PDF
%----------------------------------------------------------------------
% © 2012, CLAIRE WALTON. Some Rights Reserved.
%======================================================================
function PDF = Uniform_PDF(discretization_values,w0,wf)

PDF = ones(1,length(discretization_values))./abs(wf-w0);
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] A. Tiwari, “*Small Boat and Swarm Defense: A Gap Study*”, Thesis, Naval Postgraduate School, September 2008

[2] H. Chung, E. Polak, J. O. Royset, and S. Sastry, “*On the Optimal Detection of an Underwater Intruder in a Channel Using Unmanned Underwater Vehicles*”, Naval Research Logistics, Volume 58, Issue 8, pages 804–820, December 2011

[3] J. O. Royset, and H. Sato, “*Route Optimization for Multiple Searchers*”, Naval Research Logistics, Volume 57, Issue 8, pages 701–717, December 2010

[4] J. S. Jang, and C. J. Tomlin, “*Control Strategies in Multi-Player Pursuit and Evasion Game*”, AIAA Guidance, Navigation, and Control Conference and Exhibit, August 2005

[5] V. Shaferman, and Y. Oshman, “*Cooperative Interception in a Multi-Missile Engagement*”, AIAA Guidance, Navigation, and Control Conference, August 2009

[6] H. S. Shin, “*Study on Cooperative Missile Guidance for Area Air Defence*”, PhD Thesis, http://dspace.lib.cranfield.ac.uk/handle/1826/6934, Cranfield University, 2010

[7] N. Rozen, “*Sensor-Interceptor Operational Policy Optimization for Maritime Interdiction Missions*”, Thesis, Naval Postgraduate School, 2009

[8] R. Ghabcheloo, I. Kaminer, A. P. Aguiar, and A. Pascoal, “*A General Framework for Multiple Vehicle Time-Coordinated Path Following Control*”, American Control Conference, 2009

[9] C. Walton, “*Performance Criterion for Multiple Agent Interception Scenarios*”, University of California, Santa Cruz, June 21, 2012

[10] P. Zarchan, “*Tactical and Strategic Missile Guidance*”, Fifth edition, Reston, VA, American Institute of Aeronautics and Astronautics Inc., 2007

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code ME
   Department of Mechanical and Aerospace Engineering
   Naval Postgraduate School
   Monterey, California

4. Professor Isaac I. Kaminer, Code MAE/Ka
   Department of Mechanical and Aerospace Engineering
   Naval Postgraduate School
   Monterey, California

5. Associate Professor Johannes O. Royset, Code OR
   Department of Operations Research
   Naval Postgraduate School
   Monterey, California

6. Professor Yeo Tat Soon
   Temasek Defense Systems Institute
   National University of Singapore
   Singapore

7. Ms Tan Lai Poh
   Temasek Defense Systems Institute
   National University of Singapore
   Singapore

8. Mr William Lau
   CTO, DSO National Laboratories
   Singapore